Autonomous Control of Simulated Fixed Wing Aircraft using Deep Reinforcement Learning

Gordon Rennie

MSc in Computer Science
The University of Bath
September 2018

Autonomous Control of Simulated Fixed Wing Aircraft using Deep Reinforcement Learning

submitted by

Gordon Rennie

for the degree of MSc in Computer Science of the

University of Bath

September 2018

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of MSc in Computer Science in the Department of Computer Science. No portion of the work in this thesis has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signature of Author.....

Gordon Rennie

- ❑ **Retain for Sample** **Mark**

UNIVERSITY OF BATH **Department of Computer Science** **INDIVIDUAL COURSEWORK** Submission Cover Sheet **Please fill in both columns in BLOCK CAPITALS and post into the appropriate Coursework Submission Box.**	*for office use* **Date and time received** **This section will be retained by the Department Office as confirmation of hand-in**
How to present your work 1. Bind all pages of your assignment (including this submission sheet) so that all pages can be read by the marker without having to loosen or undo the binding. Ensure that the binding you use is secure. Missing pages cannot be marked. 2. If you are required to submit part of the work on a disk, place the disk in a sealed envelope and bind the envelope into the submission. You must keep a copy of your assignment and disk. The original is retained by the Department for scrutiny by External Examiners.	**Declaration** *I certify that I have read and understood the entry in the Department of Computer Science Student Handbook on Cheating and Plagiarism and that all material in this assignment is my own work, except where I have indicated with appropriate references. I agree that, in line with Regulation 15.3(e), if requested I will submit an electronic copy of this work for submission to a Plagiarism Detection Service for quality assurance purposes.*
FAMILY NAME Rennie	FAMILY NAME Rennie
GIVEN NAME Gordon	GIVEN NAME Gordon
UNIT CODE CM50170	UNIT CODE CM50170
UNIT TITLE Dissertation	UNIT TITLE Dissertation
DEADLINE TIME & DATE 17:00, Friday 14th September	DEADLINE TIME & DATE 17:00, Friday 14th September
COURSEWORK PART (if applicable)	COURSEWORK PART (if applicable)
SIGNATURE	SIGNATURE

Abstract

Autonomous control of aircraft is a challenging high-dimensional continuous control problem with applications in Unmanned Aerial Vehicles, autopilot systems and flight simulators. The problem domain appears well-suited to reinforcement learning (RL), a machine learning subfield which implements agents capable of learning from interactions with an environment. Recent advances in the application of deep neural networks to RL have allowed agents to perform well in increasingly complex tasks, including continuous control tasks.

The problem of heading and altitude control of fixed-wing aircraft is formulated in the RL framework as a Markov decision process. A new software package implementing flight control environments is developed by integrating the JSBSim flight dynamics model. The resulting software package, Gym-JSBSim, provides configurable and fast flight control environments with 3D visualisation. Gym-JSBSim conforms to the OpenAI Gym interface and is published under an open source license.

A series of experiments evaluating the performance of deep RL agents using the proximal policy optimisation algorithm is then conducted. The results demonstrate the agents are able to learn effective control policies for maintaining a target altitude and heading by directly adjusting control surface positions with continuous actions. Agents perform less well in a more complex flight environment which requires the aircraft to be turned, and further work concentrating on improved action exploration is identified.

Acknowledgements

I extend my deep gratitude to my supervisor, Dr. Özgür Şimşek, for her invaluable help and guidance throughout this project. I thank also Dr. Michael Carley for his advice regarding aircraft control and simulation packages. Finally, I thank my partner Ashleigh for her support and patience during this journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Aircraft flight control is a challenging, high-dimensional problem with applications in Unmanned Aerial Vehicles (UAVs), autopilot systems and flight simulators. A moving aircraft has a large state space comprising translation and rotation in three dimensions, and a continuous action space comprising inputs to control surfaces and the aircraft's propulsion. Conventional autopilot systems for fixed wing aircraft are commonly implemented through nested proportional-integral-derivative (PID) controllers, which require parameter tuning and are vulnerable to control instability in perturbed flight conditions (Kasnakolu, 2016).

Reinforcement learning (RL) is a field concerned with implementing agents capable of learning from interactions with their environment. When operating in environments with large state and/or action spaces, the ability of an agent to *generalise* its experience across many states and actions becomes essential to achieve good performance within practical learning times. The application of artificial neural networks for generalisation in RL agents has recently found significant success, for example surpassing human world-champion level performance in board games and video games (Silver et al., 2016; OpenAI, 2017) and learning locomotion in complex robotic control environments (Heess et al., 2017). The subfield concerning agents using deep neural networks is known as *deep reinforcement learning*.

1.2 Scope

This work investigates the application of deep RL to the heading and altitude control of fixed-wing aircraft and develops a software package to enable research in this area. Whereas previous works applying RL to aircraft control have simplified the problem by discretising the action space, reducing the number of dimensions in the state space, and/or provided human expert demonstrations, this project achieves full six degrees-of-freedom aircraft control with continuous actions and no human demonstration. The proximal policy optimisation (PPO) algorithm is applied, motivated by its recent successes in similarly complex, continuous domains such as locomotion and dexterous hand manipulation (Heess et al., 2017; Andrychowicz et al., 2018).

Significant theoretical attention is devoted to the design of reward functions for flight control, which encode the behaviour we wish the agent to learn. Useful forms of reward functions and reward shaping functions are derived, which have general applicability to continuous control tasks.

A new software package called Gym-JSBSim is developed, named after the high-fidelity flight dynamics model JSBSim that it integrates. Gym-JSBSim implements the commonly-used OpenAI Gym interface (Brockman et al., 2016) and benefits from a modular design, fast execution speed, compelling 3D visualisation options and free availability under an open source license.

Following the implementation of Gym-JSBSim, agents using the proximal policy optimisation (PPO) algorithm are evaluated in environments to control the aircraft's altitude and direction (*heading*). The results showed that agents were able to learn effective, but noisy, control policies in one of two control tasks. Similar performance was achieved whether the environment was modified to fly a light aircraft, a fighter jet or a passenger airliner, demonstrating the flexibility of RL over conventional control methods, which would require re-tuning as aircraft handling changed.

A second control task was more complex, and rewarded agents for turning the aircraft to a target heading. Agents learned to maintain their altitude, but did not turn to the desired heading, even when provided shaping rewards or in the presence of exploratory action noise. Methods to improve performance by better exploration are identified from recent literature and proposed for future investigation.

Chapter 2

Literature Survey

A review is conducted of relevant theory and previous works in RL, deep RL, and aircraft control.

2.1 Reinforcement Learning

This section will formulate the agent-environment model for RL, describe common RL approaches, review how these approaches have been augmented with deep neural networks, and explore how best to formulate tasks as RL problems. An emphasis in the review will be placed on deep RL algorithms and approaches capable of continuous action control, as required for aircraft control.

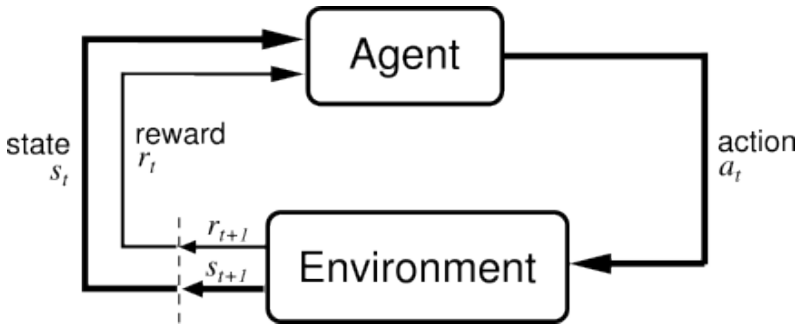

Figure 2-1: Interactions between an RL agent and its environment. (Sutton and Barto, 1998, chap.3.1)

2.1.1 The Agent-Environment Interaction

In this work we formulate the control problem in the standard way as a fully-observed Markov decision process (MDP), with an agent interacting with an environment at discrete time steps (Figure 2-1). Given the agent is in state s, it selects an action, a, and receives from the environment a new state s' with a reward signal r. We assume interactions consist of sequences of finite length known as *episodes*; an episode ends when a terminal state is reached.

We define an episodic MDP, based on a modification of the infinite-horizon MDP definition (Marthi, 2007), thus:

Definition 2.1. *An episodic Markov decision process is a tuple:*

$$\mathcal{M} = (S, A, P, R, d, T)$$

for,

S**,** *the set of states*

A**,** *the set of actions*

$P(\cdot|s, a)$**,** *the transition probability distribution upon taking action* $a \in A$ *in state* $s \in S$

$R(s, s')$**,** *the scalar reward resulting from transition* (s, s')

$d(\cdot)$**,** *the probability distribution over the initial state*

$T \subset S$**,** *the set of terminal states*

An agent implements a map of states to actions known as a *policy*, π:

$$\pi(s) = a$$

The RL agent aims to maximise its cumulative reward over each episode. However, in the general case the agent has no knowledge *a priori* of the environment transition dynamics, P, or the reward function, R. Therefore the agent learns through interactions with the environment, and applies an appropriate algorithm to improve its policy in response to experience.

This chapter will describe two categories of approaches to learning an optimal policy: value-based methods, and policy-based methods. Following this, algorithms applying deep neural networks to these approaches will be examined.

2.1.2 Value-Based Methods

Value-based methods for RL rely on a value function to estimate the expected return from a state s_t given policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\{G_t \mid s_t = s\}$$

where $\mathbb{E}_{\pi}$ is the expected value following policy π, and G_t is the cumulative reward (*return*) from step t until termination. Frequently, a more useful form is the action value function, which is the expected return for taking action a at s then following π thereafter (Watkins and Dayan, 1992):

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}_{\pi}\{G_t \mid s_t = s, a_t = a\} \\ &= \mathbb{E}_{\pi}\{r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \ldots \mid s_t = s, a_t = a\} \end{aligned}$$

where $\gamma \in [0,1]$ is a discount factor. This can be decomposed to yield a Bellman equation relating the action value of a state to the action value of the next state:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\{r_{t+1} + \gamma \cdot Q^{\pi}(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a\}$$

Every finite MDP has at least one optimum policy π^* which gives the maximum expected return (Sutton and Barto, 1998). If the corresponding optimal action value function, Q^*, is known, the optimal policy action at each state can be determined by selecting greedily from the available actions: $\max_a Q^*(s,a)$.

An agent using a value-based method aims to learn a good approximation of the optimal action value function and thus a good approximation of the optimal policy. Two widely used RL algorithms, Q-learning (Watkins and Dayan, 1992) and state-action-reward-state-action (SARSA) (Rummery and Niranjan, 1994), achieve this through *bootstrapping*: improving the estimate of $Q^{\pi}(s_t, a_t)$ using the next state's estimate $Q^{\pi}(s_{t+1}, a_{t+1})$ through an update equation with the general form:

$$Q^{\pi}(s,a) \leftarrow Q^{\pi}(s,a) + \alpha \cdot \delta$$

where $\alpha > 0$ is the *learning rate* parameter and δ is the temporal-difference

(TD) error. TD error represents the difference between the agent's previous estimate of $Q^\pi(s, a)$ and the new estimate made from interacting with the environment. It can be represented as (Arulkumaran et al., 2017):

$$\delta = Y - Q^\pi(s_{t+1}, a_{t+1})$$

where Y is the *target* value estimated by the algorithm.

SARSA is an *on-policy* algorithm which generates target values from actions chosen by the current policy:

$$Y = r_t + \gamma \cdot Q^\pi(s_{t+1}, a_{t+1})$$

Conversely, Q-learning is an *off-policy* algorithm which approximates the optimal policy by selecting the next action greedily from the available estimates of action values:

$$Y = r_t + \gamma \cdot \max_a Q^\pi(s_{t+1}, a)$$

Value estimates are improved by performing an update following every agent action, which in turn results in an improved policy. This process is guaranteed to converge to the optimal value function and policy when all states are visited an infinite number of times (Sutton and Barto, 1998, chap.6.4). In practice, an acceptable approximation is reached within a finite number of interactions.

Tabular vs. Approximated Value Functions

The value functions described so far have assumed that unique values are stored and updated for individual states or state-action pairs. This *tabular* approach is applicable only to environments with small, finite state-action spaces where it is feasible to store the tables of values and visit enough state-action pairs a sufficient number of times to converge to a good approximation.

The tabular approach becomes infeasible when the environment has a large state and/or action space, including when they are infinitely large due to the presence of continuous state or action variables. The computational time required for the agent to experience and learn from all state-action combinations becomes impractical if not impossible, as may the memory space requirements

for storing tabular values. This is an example of the *curse of dimensionality*, a concept introduced by Bellman (2015) in his work on dynamic optimisation problems.

Value function approximation is applied to counter the curse of dimensionality. In this approach, state or action values are calculated from a parameterised *value function*. Value function parameters are updated from experiences to minimise the TD error using an appropriate optimisation technique, such as gradient descent. Value function approximators with a linear form are commonly applied (Sutton and Barto, 1998, chap.8.3):

$$V_t(s) = \vec{\theta_t} \cdot \vec{\phi}(s)$$

where $\vec{\theta_t}$ is the parameter vector and $\vec{\phi}(s)$ is a vector of features extracted from state s. The parameter vector has the same length as the feature vector, and provides weights of how much each feature contributes to the value. Features can be hand-crafted by the designer using domain knowledge to identify which aspects of state are most important. Alternatively, features can be generated through methods such as tile coding, radial basis functions or Fourier basis functions (Konidaris, Osentoski and Thomas, 2011).

The popularity of linear value function approximation in RL arises from good convergence properties which permit stable learning (Tsitsiklis and Roy, 1997). Non-linear function approximators in the form of neural networks examined will be examined in Section 2.1.4; these have advantages over linear function approximators for representing more complex functions (Sanguineti and Hlaváčková-Schindler, 1999), but have no convergence guarantee and require additional techniques to stabilise the learning.

2.1.3 Policy-Based Methods

The value-based methods described in Section 2.1.2 require agents to select actions by examining a table or function of learned values (e.g. to select greedily with respect to value). Policy-based methods instead rely on learning a set of parameters, θ, for a parameterised policy which selects actions directly from its state (Sutton and Barto, 2018, p.323). Often, the action selection is probabilistic: $\pi(a \mid s, \theta)$ (Arulkumaran et al., 2017).

Action selection from a paramaterised policy has a number of advantages:

updates to policy parameters smoothly change the action selection distribution, which give better convergence properties than value-based methods (Sutton and Barto, 2018, p.326); it is easily applied to continuous action control, by selecting the action from a distribution defined by mean and standard deviation parameters (Sutton and Barto, 2018, p.337); and it avoids having to search for $\max_a Q$, which becomes a computationally expensive operation in high-dimensional action problems such as robot locomotion (Silver, 2016).

Various policy-based methods exist; here we will focus on the policy gradient and actor-critic methods that are most applicable to deep RL (Arulkumaran et al., 2017).

Policy Gradient Methods

Policy gradient algorithms are formulated for maximising an objective function $J(\theta)$. Using gradient ascent, the policy parameters θ are updated to maximise $J(\theta)$ (Sutton and Barto, 2018, p.323):

$$\theta_{t+1} = \theta_t + \alpha \cdot \nabla J(\theta_t)$$

The gradient $\nabla J(\theta_t)$ is estimated using a result from policy gradient theorem (Sutton et al., 1999):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}(\nabla_\theta \log \pi_\theta(a \mid s, \theta) \cdot Q^{\pi_\theta}(s, a)) \tag{2.1}$$

There are multiple ways to obtain an estimate of $Q^{\pi_\theta}(s, a)$. One way is to obtain actual returns from episodes of the agent interacting with the environment with policy π_θ (*Monte Carlo* sampling). In this case, $Q^{\pi_\theta}(s, a) = G$, yielding a REINFORCE algorithm variant (Williams, 1992) with the update rule:

$$\theta \leftarrow \theta + \alpha \cdot \nabla_\theta \log \pi_\theta(s_t, a_t) \cdot G$$

for all $t = 1, \ldots, T - 1$ steps in each sample episode.

REINFORCE benefits from good convergence guarantees (Phansalkar and Thathachar, 1995). A disadvantage of REINFORCE is that learning is performed off-line at the end of each episode; the agent does not benefit immediately from its experiences. Additionally, there can be high variance in

the Monte Carlo sample rewards received by the agent, particularly over long episodes. This causes noise in the learning process and reduces its speed. As a solution to this, optimisations involving discount factors and subtracting baseline values from episode rewards are possible (Weaver and Tao, 2001). The use of a baseline to improve policy gradient ascent will be detailed further in Section 2.1.4 for the asynchronous advantage actor-critic algorithm.

Actor-Critic Methods

An alternative approach to obtain estimates of $Q^{\pi_\theta}(s, a)$ for Equation 2.1 is to use value-based methods (such as Q-learning or SARSA; see Section 2.1.2) in parallel as a *critic*. The critic's purpose is to learn a value function approximation with parameters w, $Q_w(s, a) \approx Q^{\pi_\theta}(s, a)$ which will be used to train the *actor's* policy parameters, θ. (Silver, 2016)

The actor-critic formulation gains the benefits of on-line learning from every agent interaction. Two updates occur each interaction: one to improve the actor's θ and one to improve the critic's w (Silver, 2016):

$$\theta \leftarrow \theta + \alpha \cdot \nabla_\theta \log \pi_\theta(s, a) \cdot G$$
$$w \leftarrow w + \alpha^w \cdot \delta \cdot \phi(s, a)$$

2.1.4 Deep Reinforcement Learning

The augmentation of reinforcement learning with deep neural networks (NNs) in recent years has led to the successful application of RL to new and more complex problem environments (Arulkumaran et al., 2017). The use of NNs for function approximation will be described, before detailing several state-of-the-art *deep reinforcement learning* algorithms.

Neural Networks for Function Approximation

Artificial NNs are composed of connected nodes arranged in input, hidden, and output layers. Transmission occurs between nodes in each layer according to each connection's *weight*. A node sums signals received from the previous layer, processes them according to its *activation function*, and transmits its output to the next layer.

NNs are *universal approximators*: they are able to represent a wide variety of continuous functions (Hornik, 1991). The method of backpropagation is

used to adjusted the NN's weight parameters to match the desired output by minimising a *loss function*. In RL, NNs can be trained using the mean squared error of TD error, δ, as the loss function, resulting in NNs which map state inputs to the value of the value function at that state (Silver, 2016).

An early breakthrough in the application of NNs to RL for function approximation was TD-Gammon, an RL agent trained to play backgammon using a neural network for value approximation in 1992 (Tesauro, 1995). Its NN featured a single hidden layer of 80 neurons taught by the TD(λ) value-based algorithm, and reached parity with expert human players (Tesauro, 1995).

Despite the excitement surrounding the use of NNs by TD-Gammon, further successes did not quickly follow in other domains (Pollack and Blair, 1996). It is now understood that the nature of backgammon gives it a smooth value function which allowed the NN to stably train by backpropagation (Silver, 2015). In other applications, agents failed to learn due to the poor stability of NN backpropagation (Tsitsiklis and Roy, 1997).

Deep NNs comprising many hidden layers have gained great interest in recent years, driven by successes in supervised learning. Additional layers of neurons, in combination with correct selection of activation functions, permit greater accuracy in approximation and classification tasks (Lecun, Bengio and Hinton, 2015).

Applying deep NNs to RL algorithms has recently been enabled by innovations which stabilise backpropagation; these shall now be examined.

Deep Q Networks

Deep RL was enabled by two key implementation details applied by Mnih et. al (2015). Firstly, an *experience replay* memory bank of agent experiences, $e_t = (s_t, a_t, r_t, s_{t+1})$ is stored, from which experiences are randomly sampled in batches for learning. This allows the NN to learn from relatively independent and uncorrelated samples, without which training would become unstable and may diverge. Secondly, a frozen set of value approximation parameters θ^- is maintained for calculating target Q values during updates. This prevents instabilities which commonly arise when positive updates are made to $Q(s_t, a_t)$ which increase $Q(s_{t+1}, a)$ for all a, leading to oscillations and divergence. (Mnih et al., 2015)

The resulting algorithm, called Deep Q-Networks (DQN) uses a mean

squared error loss function $L_i(\theta_i)$ to perform NN backpropagation updates:

$$L_i(\theta_i) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})}\left[\left(r_t + \gamma \cdot \max_{a_{t+1}} Q^{\theta^-}(s_{t+1}, a_{t+1}) - Q^{\theta}(s_t, a_t)\right)^2\right]$$

DQN demonstrated robust stability and successful learning across a range of Atari games. The agent received state from the environment as a set of raw RGB image frames from the game and extracted features from them using convolutional layers, which represented a significant advance in the state-of-the-art. A drawback of the experience replay is that it prevents DQN from being used with eligibility trace methods (Silver, 2015), however it was necessary to stabilise learning.

A number of improvements have since been applied to the initial DQN implementation. Gu et al. (2016) extended DQN to continuous control problems using a normalised advantage function (NAF). A positive error bias introduced by the $\max_a$ operation in target value calculation was mitigated by using different parameter sets, θ and frozen θ^-, to select and evaluate actions during updates in a technique known as Double DQN (Hasselt, Guez and Silver, 2016). Prioritising the experience replay order by TD error was shown to improve performance by Schaul et al. (2015). Finally, Wang et al. (2016) separated the action value function into two components in their Dueling Network architecture:

$$Q(s, a) = V(s) + A(s, a) \tag{2.2}$$

In the Dueling Network approach, two separate networks are trained: one to estimate the value of the current state $V(s)$ and another to calculate the advantage of taking a given action in that state $A(s, a)$. Separating these components allows the agent to better discriminate between similarly-valued actions and improves learning efficiency. An agent implementing prioritised replay, double DQN and dueling networks was able to achieve a mean score three times higher across the Atari games suite than the original DQN (Silver, 2016).

Asynchronous Advantage Actor-Critic

The asynchronous advantage actor-critic (A3C) is a policy-based actor-critic algorithm which uses two deep NNs as function approximators for the actor

and critic (Mnih et al., 2016).

A3C's actor updates are improved by using an estimate of the state's value, $V^w(s)$, as a baseline to reduce variance. Updates are performed every n steps or when the episode terminates, by firstly estimating Q from the n-step sample using an estimate of V^w from the critic:

$$Q(s_t, a_t) = r_{t+1} + \gamma \cdot r_{t+2} + \cdots + \gamma^{n-1} \cdot r_{t+n} + \gamma^n \cdot V(s_{t+n})$$

The actor update is then carried out on the action advantage, $A(s, a)$ (Equation 2.2):

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\big(\nabla_\theta \log \pi_\theta(a \mid s, \theta) \cdot (Q(s, a) - V^w(s))\big) \\ &= \mathbb{E}_{\pi_\theta}\big(\nabla_\theta \log \pi_\theta(a \mid s, \theta) \cdot (A(s, a))\big) \end{aligned}$$

The critic network is then updated, using the mean squared error of the advantage as the loss function for backpropagation:

$$L(w) = \big(Q(s, a) - V^w(s)\big)^2$$

The described approach is the advantage actor critic. A final innovation was to stabilise backpropagation in the NNs through an asynchronous implementation of the algorithm: many agent *workers* operate in parallel to generate experience, which is used to train globally shared actor-critic NNs. This generates relatively independent experiences from many workers, which allows robust learning without an experience replay buffer (Mnih et al., 2016). This asynchronous element gives A3C its full name.

A3C was demonstrated to achieve efficient learning and good policies in a range of continuous locomotion control tasks. It outperformed the original DQN implementation on the Atari environment suite by a factor of four based on mean performance across games (Silver, 2016).

Proximal Policy Optimisation

An important strategy for improving the stability of deep policy-based methods is to reduce the size of the policy change resulting from an actor parameter update. The proximal policy optimisation (PPO) algorithm (Schulman et al., 2017) applies this concept by introducing a modified loss function with clip-

ping. Its clipping parameter, ϵ, becomes an agent hyperparameter which limits the extent that an action probability can change during a gradient descent update.

The good stability and sample efficiency of PPO makes it a common choice in continuous control domains, e.g. robotic locomotion (Heess et al., 2017) and dexterous manipulation (Andrychowicz et al., 2018).

2.1.5 Reward Functions

Deep reinforcement learning algorithms which allow an agent to learn near-optimal policies that maximise reward in complex control environments have been described. However, specifying the reward function which encodes the agent's goals is recognised as a highly challenging task itself (Regan and Boutilier, 2012). Additionally, selecting a sparse reward function which does not guide the agent to desirable intermediate states may result in infeasibly long learning time. This section will examine how to design reward functions which correctly encode the desired behaviour and greatly improve the speed with which agents converge to optimal behaviour.

Reward Function Design

The reward function, R, maps state transitions to a real-valued scalar reward, r:

$$R(s, s') = r$$

The reward function implicitly defines the optimal policy which an RL agent will attempt to learn (Marthi, 2007). Regan and Boutilier (2012) note that specifying an appropriate reward function is a cognitively demanding and time-consuming process which often requires domain-specific expertise. They attribute the difficulty of reward function design partially to humans' poor ability to precisely quantify their preferences in utility functions.

RL practitioners have commonly reported reward functions inadvertently causing unwanted behaviour in agents. For example, agents may use undesirable behaviours to achieve their goal (as when a robotic arm throws objects instead of placing them as intended; see Hammond, 2017) or repeatedly perform rewarding intermediate actions without progressing to the intended goal

(Randløv and Alstrøm, 1998; Clark and Amodei, 2016). Story (2017) summarises several reward function design considerations:

- An agent receiving positive rewards from the environment learns to repeat the most rewarding behaviours. Such an agent will avoid terminal states, unless transitioning to the terminal state provides an even larger reward than the agent would accumulate otherwise.

- An agent receiving only negative rewards (*e.g.* -1 every timestep) is incentivised to minimise the number of timesteps in the episode. The agent will learn to reach terminal states as fast as possible, unless the terminal provides a large negative reward.

These concepts were developed formally by Dewey (2014) who described the conditions for *dominant* policies to emerge. Such policies will always receive a higher reward than a *dominated* policy, no matter what terminal rewards are scheduled by the human designer. Dewey argues that the number of dominated policies increases with agent and environment complexity, which make it more difficult to formulate an effective reward function.

In summary, reward functions must be carefully crafted to elicit the desired behaviour, with particular attention rewards on terminal conditions which may allow agents to exploit the system in unintended ways.

Reward Shaping

A challenge in RL arises when agents explore large state-action spaces in environments which provide *sparse* rewards (Figure 2-2a). With a sparse reward, agents receive no reward feedback to improve their policy until they happen to explore the rewarding state by chance, which may require impractical training times (Randløv and Alstrøm, 1998). A solution to this is *reward shaping*, a method where intermediate rewards are given to the agent to guide it in the direction of its goals.

A shaping reward can be constructed as the sum of the environment's usual reward, R, plus a shaping reward function, F:

$$\hat{R}(s, s') = R(s, s') + F(s, s')$$

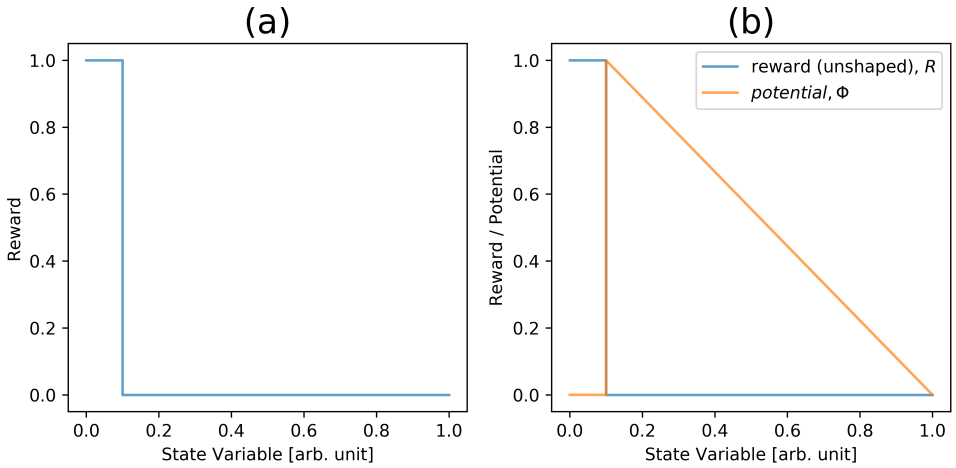

Figure 2-2: Reward and potential against an arbitrary state variable for **(a)** a sparse reward function and **(b)** a potential based shaping reward function. It is assumed the goal states s_G are terminal, with $\Phi(s_G) = 0$.

F is specified to provide useful feedback to the agent, *e.g.* positive reward when the agent moves closer to a goal state, and negative reward when it moves further away.

Early works in reward shaping identified "bugs" which could arise when using the technique. Randløv and Alstrøm (1998) found that a bicycle-riding agent given a shaping reward based on its progress to a goal learned to cycle in a tight circle around the goal without ever reaching it. Ng, Harada and Russell (1999) studied the phenomenon, and reasoned the problem arose when an agent could repeatedly cycle through a sequence of states $(s_1, s_2, \ldots, s_n, s_1)$ while receiving a net positive reward.

To avoid this problem, formally we want all optimal policies of an MDP $M' = (S, A, P, R + F, d, T)$ using reward shaping to be optimal in the corresponding non-shaping reward MDP $M = (S, A, P, R, d, T)$. We call this property *policy invariance*. Ng, Harada and Russell (1999) proved that policy invariance is achieved in the general case if and only if the shaping function F is *potential-based*[1]:

$$F(s, s') = \Phi(s') - \Phi(s)$$

[1] equation shown is for the undiscounted case

where $\Phi : S \to \mathbb{R}$ is the *potential function*. In an episodic MDP we have the additional requirement that $\Phi(s_T) = 0$ for all terminal states $s_T \in T$ to preserve policy invariance (Grześ, 2017).

The potential function encodes domain knowledge of how "good" a given state is. The shaping reward then provides the agent frequent feedback on whether it is improving or worsening its state, as measured by $\Delta\Phi$ (Figure 2-2b). There is a strong theoretical basis to the notion of telling the agent how good a state is with the shaping reward function: Wiewiora (2003) showed that a potential-based shaping reward function is equivalent to initialising a value-based RL agent with particular pre-set Q values.

In some domains it may be advantageous to give agents shaping rewards dependent on multiple state variables. Story (2017) gives the example of the LunarLander environment, in which an agent must descend a lander to the moon's surface and touch-down at a safe velocity using actions on its thrusters. In this environment, providing a shaping reward for descent to the surface results in a large number of crashes and poor reward. A better shaping reward function depends on both proximity to the surface and velocity, which will encourage the agent to meet both requirements.

2.2 Aircraft Flight Control

2.2.1 Aircraft State

The state of an aircraft can be represented with various references and coordinate systems. I propose to use the same body-framed coordinates (Figure 2-3) applied by Kim et al. (2004) for helicopter RL control:

$$s_b = \{\phi, \theta, \dot{x_b}, \dot{y_b}, \dot{z_b}, \dot{\phi}, \dot{\theta}, \dot{\omega}\}$$

where the parameters correspond to orientation (ϕ pitch, θ roll, ω yaw), velocity ($\dot{x_b}$, $\dot{y_b}$, $\dot{z_b}$) and angular velocity ($\dot{\phi}$, $\dot{\theta}$, $\dot{\omega}$). This benefits from embedding the symmetry of the problem domain into the representation[2].

In this work the agents will be given full and perfect information of their state.

[2] with a spatial (Earth) frame of reference, the agent would have to learn that flying north at an arbitrary coordinate has much the same dynamics as flying east at another

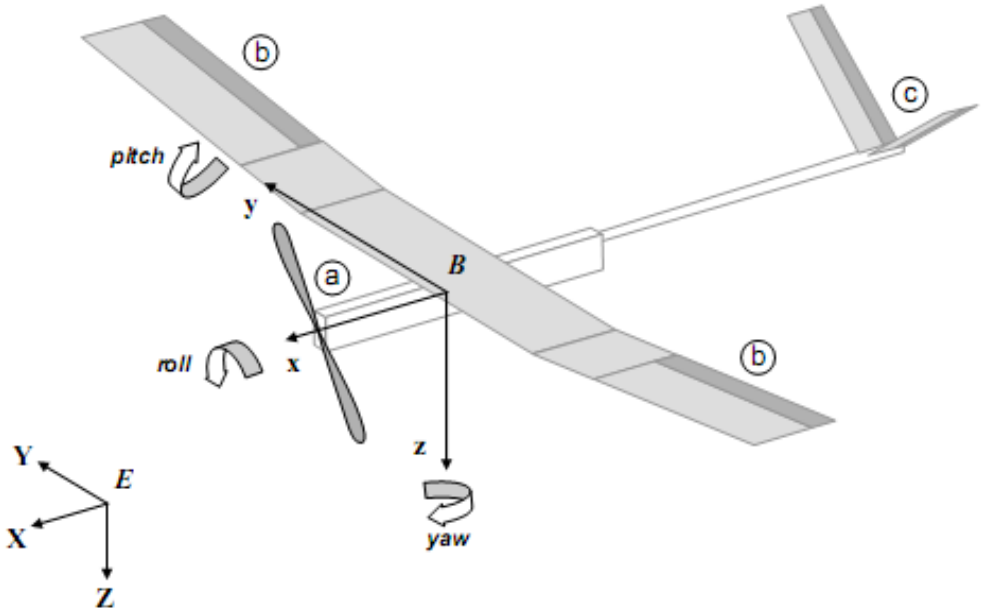

Figure 2-3: Aircraft body axes. (Kimathi, Kang'ethe and Kihato, 2017)

The state representation will be expanded with additional parameters particular to a given task. For example, in waypoint following additional terms x, y, z representing the relative location of the waypoint may be added.

2.2.2 Aircraft Controls

The actions available to the agent are to command each of the aircraft's control surfaces (Figure 2-4) to a position in the continuous range $[-1, 1]$, and its engine throttle in $[0, 1]$:

$$a = \{\delta_a, \delta_e, \delta_r, \delta_t\} \in [-1, 1]^3 \times [0, 1]$$

where δ_a is aileron deflection, δ_e is elevator deflection, δ_r is rudder deflection and δ_t is throttle position. The actions described here are for a subset of a conventional aircraft's controls and could easily be expanded, *e.g.* for landing gear, flaps or unconventional aircraft control surfaces.

2.2.3 Conventional Flight Control Methods

Conventional flight control autopilot systems typically comprise an architecture of nested proportional-integral-derivative (PID) controllers. Each PID

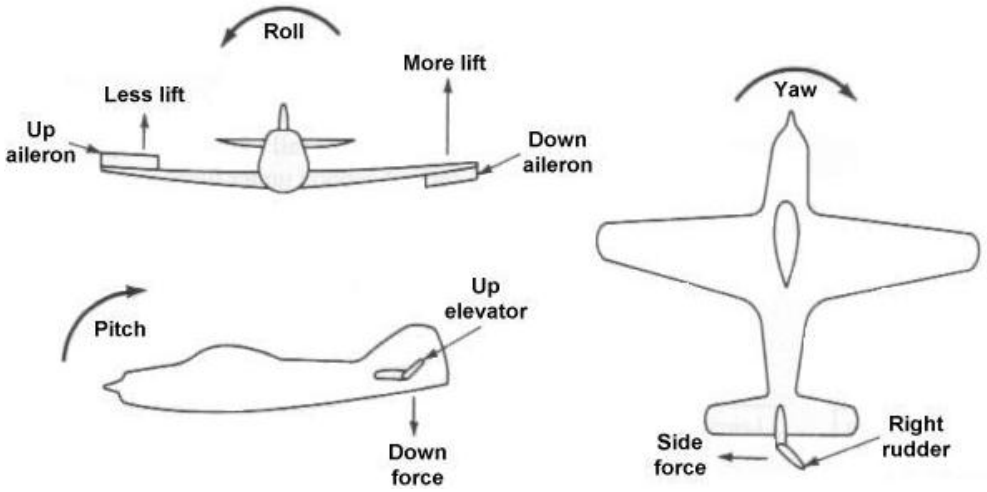

Figure 2-4: Aircraft control surfaces. (Kimathi, Kang'ethe and Kihato, 2017)

controller receives one element of the aircraft's state, such as velocity or pitch, as input and attempts for maintain it at a specified *set-point* value by adjusting its outputs, which are linked to an appropriate control surface or engine controller. (Pratt, 2000)

The full set of equation of motion for an aircraft comprise 12 nonlinear ordinary differential equations. Due to their complexity, it is common to linearise and decouple the equations to permit the design of a simpler system of PID controllers (Beard, 2012, p.60). The drawback of this approach is that the linearised equations do not hold under disturbed flight conditions. During situations such as high crosswinds or aerobatic manoeuvres, the decoupling assumptions can break down potentially leading to departure from controlled flight (Kasnakolu, 2016). Additionally, the PID controllers require manual effort to tune their parameters for good flight performance (Erdos and Watkins, 2008).

In the nested control architecture, high-level navigation controllers plan the aircraft trajectory to reach a desired location. The output of these controllers are fed to the inputs of controllers for longitudinal and lateral control of the aircraft, which actuate the aircraft's control surfaces to follow the commanded route (Pratt, 2000). Navigation controller path-following algorithms should provide useful algorithms for implementing shaped RL reward functions (see *e.g.* Beard, 2012, pp.175–183).

2.2.4 Reinforcement Learning for Flight Control

RL has been applied to the control of aircraft systems in a variety of previous works. Monaco, Ward and Barto (1998) applied a Q-learning variant to train an agent to return fixed wing aircraft in steep dives to steady, level flight. The authors aspired to achieve continuous action control in all three axes, but their implementation was limited to discrete action control in one axis (pitch).

Q-learning was applied to controlling a quadcopter in the work of Hayes (2013). The action space in this work was coarsely discretised to the values $(-1, 0, 1)$ and the state space was limited to the orientation of the quadcopter, with no control attempted over its position. Despite these concessions to reduce the dimensionality of the problem, slow learning limited the performance of the agent and caused it to under-perform compared to a conventional PID control system.

Kimathi, Kang'ethe and Kihato (2017) used a policy gradient algorithm to control the heading of a fixed wing UAV in the X-Plane simulator. The velocity and pitch of the aircraft was controlled by conventional controllers, reducing the dimensions of the problem to actions modifying the aircraft aileron, and state observing the aircraft's heading. The algorithm performed well and demonstrated superior performance over a PID controller by reaching set-point faster and with less overshoot.

The greatest success of RL in aircraft control comes from the work of Kim et al. (2004), who successfully applied the PEGASUS policy search algorithm to train an agent to hover an autonomous helicopter. The algorithm was simplified by handcrafting features based on domain knowledge. Training was initialised on experiences generated by a human pilot, before carrying out further learning in a simulation environment. The agent could be commanded to hover at an arbitrary coordinate and orientation, receiving quadratically higher reward as it moved towards the target state. The agent was successful at controlling the real-world helicopter and, by providing the agent a sequence of target positions and orientations, could follow trajectories and perform aerobatic manoeuvres.

The application of deep RL methods to flight control has so far been limited to high-level control in multi-agent systems or quadcopters. Conde, Llata and Torre-Ferrero (2017) successfully applied DQN to control groups of fixed wing UAVs flying in formation. The simulations state was simplified to two di-

mensions (x and y axes) and the action space was coarsely discretised. Polvara et al. (2017) demonstrated the use of DQN for control of a quadcopter landing. One DQN agent was used to identify a marked landing site, and a second agent using double DQN controlled the vertical descent of the UAV onto the site. The agents were trained in simulation and demonstrated good real-world performance, successfully generalising across different terrain. Agent performance was negatively impacted in real-world windy conditions, which it had not experienced in the simulator. This work reflects a growing interest in agents capable of acting from raw video inputs enabled by deep RL (Carrio et al., 2017).

In the described works by Conde, Llata and Torre-Ferrero and Polvara et al., the deep RL agents did not directly control aircraft surfaces and instead provided a set-point to an inner PID loop which executed the commands.

This literature review of related works reveals a gap in the application of RL to fixed wing aircraft control. Firstly, many previous works have observed and acted on a subset of the state-action space. This reduces the complexity of the problem, but limits the performance and applicability of the final agent. Additionally, many of these works have discretised the action space, often coarsely, to achieve feasible learning times. Even with these concessions, learning speed has been noted as a challenge. This suggests flight control is a difficult application for RL and may be well-suited to the powerful generalisation capabilities of the deep neural networks used in deep RL.

Secondly, the autonomous helicopter agent implemented by Kim et al. (2004) required significant human input, including hand-crafting of features and human expert demonstration. Additionally, this agent learned only to hover in place, and could move by being commanded to hover at a sequence of positions. It would be desirable to achieve movement and path-following behaviour with limited hand-crafting of the algorithm and without requiring an external trajectory controller.

2.3 Technology Review

A flight control RL environment requires a suitable flight simulator to provide state transitions, with the primary criteria being speed of execution and simulation accuracy. Various simulators have been applied in fixed-wing UAV research, including X-Plane, Microsoft Flight Simulator, and FlightGear

(Gimenes et al., 2008). A review of previous works and RL libraries found no existing RL environments for fixed-wing aircraft, therefore it was decided to implement one as part of this project.

At the heart of a flight simulator is its flight dynamics model (FDM), a physics engine performing numerical calculations to determine the aircraft's motion over time. It was desired to interface with an FDM directly for the RL environment to maximise computational efficiency, noting that the additional features of flight simulators (primarily graphical visualisation) are not necessary during agent training.

After careful consideration the open-source JSBSim FDM (Berndt and De Marco, 2009) from the FlightGear simulator was selected for the environment. JSBSim has the following benefits:

- Lightweight, computationally fast FDM written in C++
- Open-source and free
- Installable as a library with a Python API, allowing direct software integration into the environment without requiring network communication
- Can be run faster than real-time to allow quick training times.
- Provides a large variety of aircraft models
- Simulation state may be output to the FlightGear simulator for 3D graphical visualisation of the aircraft under control

Chapter 3

Aircraft Control as a Markov Decision Process

It is desired to control aircraft to fly steady trajectories along a specified compass direction (*heading*) while maintaining constant altitude. Two classes of episodic MDPs with increasing difficulty will be formulated, with appropriate state space, action space, initial conditions, termination conditions, and reward function. Particular attention is paid to deriving reward functions suitable for continuous control tasks, including policy-invariant shaping reward functions. The MDPs described will then be implemented in a modular software package, described in the following chapter.

3.1 Action Space

The agent is given control over the (normalised) position of its ailerons, rudder and elevator (see Figure 2-4) by allowing it to input commanded positions, c:

$$a = \{c_a, c_e, c_r\} \in [-1, 1]^3$$

where c denotes a commanded normalised position, subscript a denotes aileron, subscript e denotes elevator, and subscript r denotes rudder.

Engine controls such as throttle and fuel mixture settings are fixed at constant values with no control provided to the agent.

3.2 State Space

The aircraft's state representation is constructed by considering what information is essential to an agent to achieve the desired behaviour and preserve the Markov property. Firstly, the agent is provided knowledge of the aircraft's position, orientation and velocities using the body-framed system described in Section 2.2.1:

$$s_1 = \{h, \phi, \theta, \dot{x_b}, \dot{y_b}, \dot{z_b}, \dot{\phi}, \dot{\theta}, \dot{\omega}\}$$

where the parameters correspond to altitude (h), orientation (ϕ pitch, θ roll, ω yaw), linear velocity ($\dot{x_b}$, $\dot{y_b}$, $\dot{z_b}$) and angular velocity ($\dot{\phi}$, $\dot{\theta}$, $\dot{\omega}$).

The agent must also be provided information on the position of the aircraft's control surfaces. When an agent commands a movement of a control surface to a given position (c) the effect on the actual position (δ) is not instantaneous. Therefore it is necessary to provide the agent information on the current control surface positions at each timestep to preserve the Markov property:

$$s_2 = \{\delta_a, \delta_e, \delta_r\}$$

Finally, the agent is provided information on how close it is to the desired state. It is desired for agents to maintain an arbitrary target heading ψ_T and target altitude h_T. Providing the agent with additional policy-invariant shaping rewards based on keeping its wings level (roll, $\phi = 0$) and sideslip zero (sideslip, $\beta = 0$) to assist with flying on a constant heading will also be investigated. Therefore the agent is provided with error signals e_α for each controlled variable α, equal to the difference between the current variable value and the desired value:

$$s_3 = \{e_\psi, e_h, e_\phi, e_\beta\}$$

where e_ψ is the error between the current and target heading, e_h is the error between current and target altitude, e_ϕ is roll error and e_β is sideslip error.

Finally, since an episodic MDP is used with a terminal time condition, the remaining timesteps in the episode, t_R, is included in the state space to preserve the Markov property (Pardo et al., 2017).

Combining the state variables for aircraft state, control surfaces state,

errors, and remaining timesteps gives the full state space representation:

$$\begin{aligned} s &= s_1 \cup s_2 \cup s_3 \cup \{t_R\} \\ &= \{h, \phi, \theta, \dot{x_b}, \dot{y_b}, \dot{z_b}, \dot{\phi}, \dot{\theta}, \dot{\omega}, \delta_a, \delta_e, \delta_r, e_\psi, e_h, e_\phi, e_\beta, t_R\} \end{aligned}$$

3.3 Transition Dynamics

The MDP transition dynamics are primarily determined by the flight dynamics model, with some secondary calculations of error state variables e and remaining timesteps t_R outside of JSBSim using basic functions. The transition dynamics are approximately deterministic, within the limits of negligible rounding and convergence errors in JSBSim's calculations.

A variety of aircraft can be selected for simulation in JSBSim; when an alternative aircraft is selected the transition dynamics are altered, but all other components of the MDP remain the same.

3.4 Initial State

Two classes of MDP with varying difficulty are created by varying the initial state of the agent.

Table 3.1 lists the common values used by both classes: the aircraft begins level at 5000 ft with moderate forward velocity, zero normal and rotational velocities, and zero control surface input. The target altitude h_T is equal to the initial altitude.

The MDPs differ by their initial heading:

- for the *HeadingControl* class of MDPs, the aircraft begins on the target heading. For optimal behaviour it must maintain its current altitude and heading.
- for the more difficult *TurnHeadingControl* MDPs, the aircraft begins on a uniformly random heading in $[0°, 360°]$. An optimal policy requires the aircraft to turn to face the target heading while maintaining its altitude

The target altitude and target heading are not directly provided to the agent, but are implicitly observed through the altitude and heading error variables with the motivation that the policies learned should be more general.

Table 3.1: Initial state values common to the HeadingControl and TurnHeadingControl MDP classes.

State Variable	Symbol	Initial Value	
altitude	h	5000	ft
forward velocity	$\dot{x_b}$	cruise[a]	ft/ sec
normal velocities	$\dot{y_b}, \dot{z_b}$	0	ft/ sec
roll, pitch orientation	ϕ, θ	0	°
angular velocities	$\dot{\phi}, \dot{\theta}, \dot{\omega}$	0	°
control surface positions	$\delta_a, \delta_e, \delta_r$	0	–
altitude error	e_ϕ	0	ft
roll error	e_ϕ	0	°
sideslip error	e_β	0	°

[a] forward velocity is initialised to an approximate cruise velocity according to the aircraft flown.

3.5 Terminal States

Two conditions are assessed to determine if a state is terminal:

- if timesteps remaining $t_R = 0$, the state is terminal
- if the magnitude of the altitude error exceeds a critical value, $|e_h| > e_{h,\max}$, the state is terminal

The maximum permitted altitude deviation was set to $e_{h,\max} = 1000\,\text{ft}$. The primary motivation for this limit was to prevent aircraft entering a steep dive and achieving negative altitude, which caused undefined behaviour and crashes in JSBSim. Such exploitation of bugs in the simulation environment by agents has been observed in other works (Story, 2017). As a secondary effect, it also terminated episodes which were obviously deviating too far from the desired state, thereby concentrating exploration on good states.

3.6 Reward Function Design

3.6.1 Desired Behaviour

A reward function will be constructed that encodes the following behaviour:

- maintain aircraft altitude h at target altitude h_T, such that $e_h = h - h_T = 0$

- fly the aircraft on the target heading[1], such that $e_\psi = \psi - \psi_T = 0$

3.6.2 Policy-Defining Reward Function

A simple reward function which encodes the desired behaviour can be constructed as the negative sum of absolute errors:

$$R(s) = -\big(|e_h(s)| + |e_\psi(s)|\big). \tag{3.1}$$

for,

$e_h(s)$, the altitude error between h and h_T in state s

$e_\psi(s)$, the heading error between ψ and ψ_T in state s

The reward function in Equation 3.1 is similar to the form used by Kim et al. (2004) for helicopter control, except that they used squared error terms $(x - x_T)^2$ over the helicopter's desired hover position and orientation.

However, both of these forms are problematic for use as reward functions: arbitrary choices in the units of h and ϕ could result in one term becoming dominant and make agents insensitive to errors in the other term. Additionally, it is advantageous to provide agents rewards normalised in *e.g.* [-1,1] to improve the stability of neural network convergence (Story, 2017).

To address these problems, normalised error terms are used instead. Normalisation is complicated by unbounded variables, such as altitude h, which require more careful selection of the normalisation function. The normalised error for an arbitrary variable i, denoted $\bar{e}_i$, is therefore calculated by:

$$\bar{e}_i(s) = \frac{\frac{|e_i(s)|}{k}}{1 + \frac{|e_i(s)|}{k}} \tag{3.2}$$

where k is a scaling factor.

Equation 3.2 (illustrated in Figure 3-1) has several attractive features that led to its selection:

- $\bar{e}_i(s)$ is normalised in the range [0,1]

[1] Formally, the compass direction the aircraft faces is its *heading* and the compass direction of its velocity vector is its *track*. This work will control the track of the aircraft, but both are referred to informally as heading.

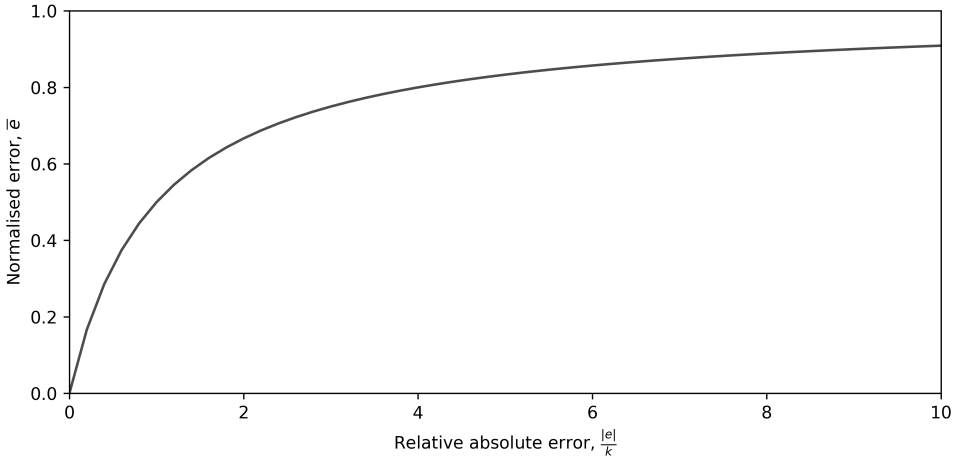

Figure 3-1: The shape of the normalised error function, showing $\bar{e} = 0$ when $|e| = 0$, and $\bar{e}$ asymptotically approaching 1 with large $|e|$.

- $\bar{e}_i(s)$ increases monotonically with $|e_i(s)|$, providing agents with smoothly varying feedback on their state
- in the event that $|e_i(s)|$ is unbounded, a large $|e_i(s)|$ causes $\bar{e}_i(s) \to 1$
- $\bar{e}_i(s) = 0.5$ when $|e_i(s)| = k$. Thus, the scaling factor k can set to an appropriate value according to the units of measure of $|e_i(s)|$, and its value can intuitively be interpreted by the designer as "at what absolute error value do I consider the agent's work to be half complete?".
- there is a relatively steep gradient at the origin, which will encourage the agent to be precise at achieving zero error (Story, 2017)

A new reward function based on the normalised error can then be defined:

$$R(s) = -\frac{1}{2}\Big(\bar{e}_h(s) + \bar{e}_\psi(s)\Big) \tag{3.3}$$

with a divisor of 2 to keep the range of $R(s)$ in $[-1, 0]$. When the agent is on the desired altitude and heading, such that $\bar{e}_h, \bar{e}_\psi = 0$ then we have $R(s) = 0$. As the errors become large, $R(s) \to -1$ to give the agent negative feedback.

As a final optimisation of the reward function, it is preferred to use positive rewards in the range $[0, 1]$ to avoid complications involving that agent exploiting terminal states (described in Section 2.1.5). To achieve this, Equation 3.3 is modified to use the complement of the normalised errors, $1 - \bar{e}$:

$$R(s) = \frac{(1 - \bar{e}_h) + (1 - \bar{e}_\psi)}{2} \tag{3.4}$$

Equation 3.4 is the final form of R used in the heading control MDPs for this work. During development, it was convenient to investigate different reward functions using the general form:

$$R(s) = \frac{1}{N} \sum (1 - \bar{e}(s)) \tag{3.5}$$

for N, the number of state variables being controlled and $\sum (1 - \bar{e}(s))$, the sum of the error complements of those variables. This general form may prove useful for formulating reward functions for other flight control tasks and other continuous control domains.

3.6.3 Additional Reward Shaping

A policy-defining reward function has been formulated (Equation 3.4), which provides the agent smoothly varying feedback on its state. However, as discussed in Section 2.1.5, complex control tasks can still benefit from additional shaping rewards in variables not covered by the base reward function to guide the agent's exploration of the state space.

To improve agent performance and/or learning speed, additional feedback is given to the agent in form of a policy-invariant potential-based shaping reward, F. State potentials are obtained from normalised error complement terms:

$$\Phi_i(s) = \begin{cases} 1 - \bar{e}_i & \text{if } s \text{ is non-terminal} \\ 0 & \text{if } s \text{ is terminal} \end{cases}$$

for $\bar{e}_i$, the normalised error of an arbitrary state variable i.

The shaping rewards passed to the agent, $\hat{R} = R + F$ can use a policy-invariant shaping function, F of the general form:

$$F(s, s') = \frac{1}{M} \sum \Phi(s') - \Phi(s) \tag{3.6}$$

for M, the number of variables with additional shaping rewards, and $\sum \Phi(s') - \Phi(s)$, the sum of those variable's potential differences. It is noted that so long as F equals the sum of potential-based components, then F itself remains a

potential-based function, as required for policy invariance.

It was noticed that agents learning heading control had rapid oscillations in their roll and yaw axes. To discourage this behaviour, a shaping reward based on two variables is given to the agent:

- keep the aircraft's wings level by keeping roll at target value $\phi_T = 0$
- minimise sideslip, β, the yaw-axis angle between the direction the aircraft's nose is pointing and its velocity vector, with target $\beta_T = 0$

These shaping rewards have a strong basis in aircraft dynamics: keeping the wings level and the aircraft's body aligned with its velocity vector reduces forces in the yaw axis, which should help the aircraft fly on a constant heading. Therefore this shaping reward provides the agent with prior domain knowledge, with the aspiration that, in addition to reducing oscillation, agents would learn policies faster and/or with better performance as a result. Equation 3.6 therefore becomes:

$$\begin{aligned} F(s, s') &= \frac{1}{2}\big(\Phi_\phi(s') + \Phi_\beta(s') - \Phi_\phi(s) + -\Phi_\beta(s)\big) \\ &= \frac{1}{2}\Big(\big(1 - \bar{e}_\phi(s')\big) + \big(1 - \bar{e}_\beta(s')\big) - \big(1 - \bar{e}_\phi(s)\big) - \big(1 - \bar{e}_\beta(s)\big)\Big) \end{aligned} \tag{3.7}$$

3.6.4 Sequential Reward Shaping

The shaping reward provided by Equation 3.7 is suitable for an aircraft to maintain a correct heading that it is already flying on. However, in the harder TurnHeadingControl MDPs, it was observed that agents initialised on an incorrect heading did not roll the aircraft to initiate a turn to the correct direction. It was theorised that this occurred because the agent was discouraged to do so by the negative shaping reward received when the wings were rolled away from the target, $\phi_T = 0$.

Inspiration was drawn from the reward shaping used by Story (2017), described in Section 2.1.5, in which agents in the LunarLander environment were encouraged to approach the landing surface, but only if they were at a safely low velocity.

It is straightforward to introduce dependency when using potentials based on normalised error complements, because they have range $[0, 1]$. We can make

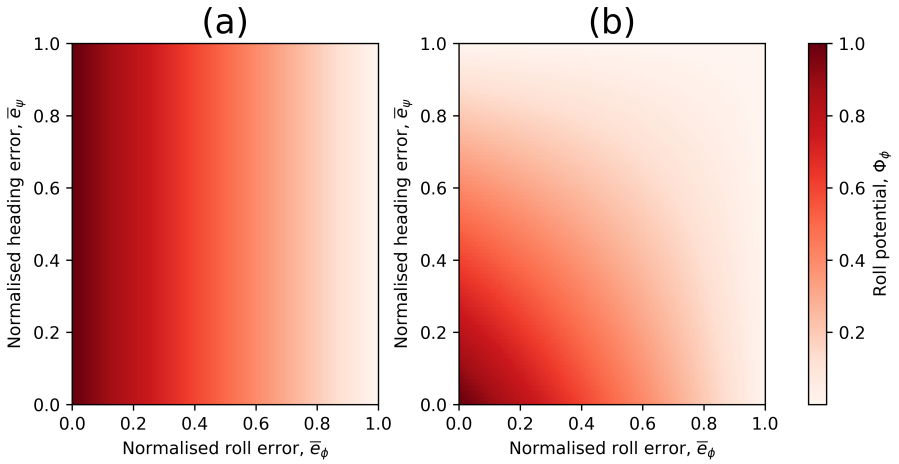

Figure 3-2: The roll potential used to calculate one contribution to the shaping reward function, F, with **(a)** no sequential dependency between roll ϕ and heading ψ, and **(b)** a sequential dependency between ϕ and ψ such that there is less potential loss (and negative shaping reward) for increasing the aircraft roll when far from the target heading. It should be easier for the agent to learn to increase its roll and turn to the correct heading using sequential potential function **(b)**.

a shaping reward on a state variable, a, dependent on a second state variable b being near its target value by multiplying their potentials:

$$\Phi_{a|b}(s) = \Phi_a(s)\Phi_b(s) \tag{3.8}$$

where $\Phi_{a|b}(s)$ is the modified potential for state a that is dependent on state variable b being near its target value. We then say that the shaping reward for a is *sequentially dependent* on b. The sequential and non-sequential potential functions are illustrated in Figure 3-2.

A variety of sequential shaping reward functions were experimented on to formulate the heading control MDPs, including those with multiple sequential dependencies. However, the final shaping reward function chosen used a shaping reward for roll, ϕ, with sequential dependence on heading, ψ, to avoid discouraging the aircraft from turning to face the correct direction. The resulting equation is:

$$
\begin{aligned}
F'(s, s') &= \frac{1}{2}\big(\Phi_{\phi|\psi}(s') + \Phi_{\beta}(s') - \Phi_{\phi|\psi}(s) - \Phi_{\beta}(s)\big) \\
&= \frac{1}{2}\big(\Phi_{\phi}(s')\Phi_{\psi}(s') + \Phi_{\beta}(s') - \Phi_{\phi}(s)\Phi_{\psi}(s) - \Phi_{\beta}(s)\big)
\end{aligned} \tag{3.9}
$$

Chapter 4

Software Engineering

This chapter describes the design and implementation details of the software developed for this work. The largest product is Gym-JSBSim, a Python module providing flight control RL environments. Gym-JSBSim integrates the JSBSim flight dynamics model (Berndt and De Marco, 2009) to provide a detailed yet fast simulation of the physical behaviour of aircraft.

Additionally, an overview of a smaller software module written to provide experimental tools such as agent training automation, hyperparameter search, and data processing is provided.

4.1 Gym-JSBSim

4.1.1 Development Processes

Development of Gym-JSBSim was managed using several modern agile software engineering practices (Sommerville, 2015, ch.9). Development was loosely organised into sprints of two weeks duration, each of which was followed by a progress review with the project supervisor. A backlog of incomplete work items was maintained, from which tasks were drawn each sprint based on priority. Automated testing, considered essential for maintaining a correct and high-quality codebase (Martin, 2008, p.121), was applied to ensure that a working product with incrementally expanding functionality was maintained throughout the development process.

4.1.2 Requirements

Gym-JSBSim was conceived to provide RL environments based on controlling simulated fixed-wing aircraft. It had the following high-level goals:

- Implement the MDPs designed in Chapter 3, providing modularity so that different tasks and reward functions can be configured for experimentation
- Provide a fast simulation with a high ratio of simulated time to real time
- Provide good compatibility with RL agent libraries commonly used in research

Requirements were captured using *user stories*, a template which captures user-oriented usage scenarios to describe required functionality and their acceptance criteria (Sommerville, 2015, p.79). New user stories were generated throughout the project in an iterative manner in response to the evolving research direction.

A summary of the requirements is provided below, numbered by their user story ID and grouped into logical categories:

Core Simulation

1 Create JSBSim instances in Python

2 Load a specified aircraft model into JSBSim

3 Set an aircraft's initial JSBSim simulation state

Agent-Environment Interaction

4 Extract state information from JSBSim

5 Input aircraft control commands to JSBSim

6 Implement the OpenAI Gym API for exchanging state/action/reward data at regular intervals with a JSBSim simulation

7 Query a Gym-JSBSim environment for its state and action space

12 Create non-learning agents which make random actions

13 Create non-learning agents which make the same, constant action

Environment and Reward Function Design

8 Experiment on a variety of control tasks as JSBSim environments in a modular framework

11 Provide a `Task` implementing aircraft heading control on the aircraft's initial heading

17 Implement a system for specifying potential-based reward functions

18 Implement a more challenging `Task` where the aircraft must turn to fly on a random heading

19 Separately track shaping rewards and assessment rewards

20 Flexibly define error-based shaping rewards on arbitrary state variables

21 Flexibly define error-based shaping rewards with sequential dependencies on arbitrary state variables

Visualisation

9 Visualise live aircraft state on a plot

10 Visualise actions on, and positions of, the aircraft control surfaces

14 Visualise live aircraft state in 3D using the FlightGear simulator

15 Visualise agent actions alongside 3D visualisations

16 Display task-related state and reward data on environment visualisations

The full Gym-JSBSim user story documentation is provided in Appendix A.1.

4.1.3 Design and Implementation

Gym-JSBSim was designed as an object-oriented program, comprising a collection of classes which encapsulate related data and behaviour. The design process involved selecting a high-priority user story and decomposing the required functionality into collaborative interactions between different classes. Design decisions were documented using Class Responsibility Collaboration (CRC) cards, which record the responsibilities and interactions of each class

Simulation	
Responsibilities	Collaborators
• instantiate and configure an instance of JSBSim (`FGFDMExec`) • get and set variables in JSBSim instance • run JSBSim simulation steps	FGFDMExec JsbSimEnv

Figure 4-1: Class Responsibility Collaboration card for the `Simulation` class.

(McLaughlin, Pollice and West, 2006, p.564). An example CRC card is shown in Figure 4-1; for the full set of CRC documentation refer to Appendix B.1.

Designs were reviewed and iterated upon before their implementation to improve compliance with object-oriented design principles, such as encapsulation and the single responsibility principle (McLaughlin, Pollice and West, 2006, ch.8). These principles helped ensure the codebase was maintainable, extensible and testable, which was important for responding to newly identified requirements and enabled the package to efficiently grow to over 4,000 lines of code.

The coding style followed clean code practices (Martin, 2008) including:

- adhering to a consistent code style format
- using variable and class names which clearly signaled intent
- favouring the use of multiple functions with narrow scope over long individual functions
- codebase version control using Git
- comprehensive automated testing, performed before pushing changes to the version control repository

This section will describe the design and inter-operation of several key classes in Gym-JSBSim.

Environment Interactions

Figure 4-2 illustrates interactions between classes during an agent action step. The `JsbSimEnv` class implements the OpenAI Gym interface by inheriting its `gym.Env` class, and receives all agent calls.

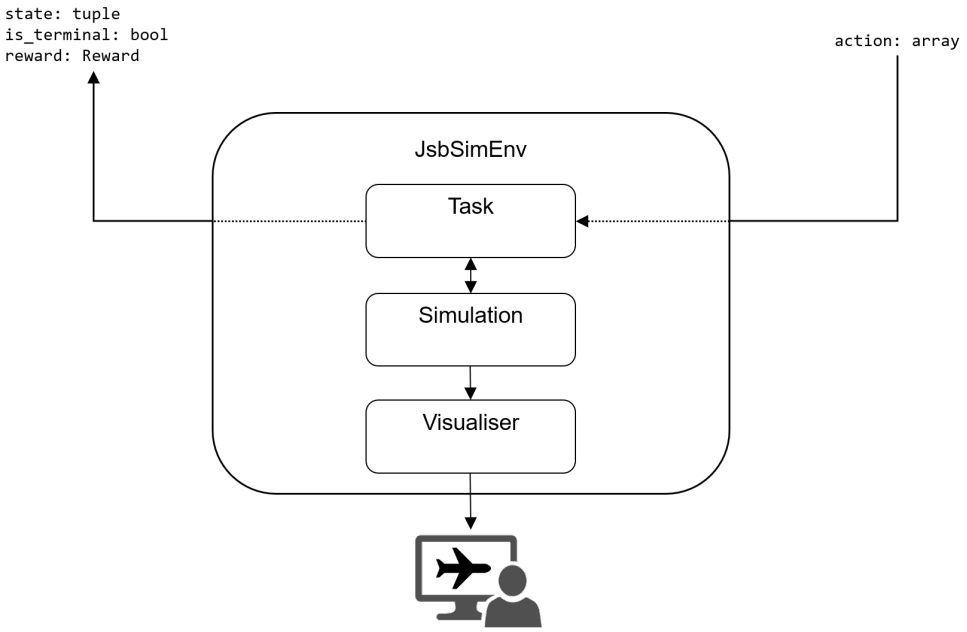

Figure 4-2: Schematic of data flow in a JSBSim environment interaction, where the environment is passed an action and returns information on state, terminality, and step reward, and may optionally output a visualisation to the user.

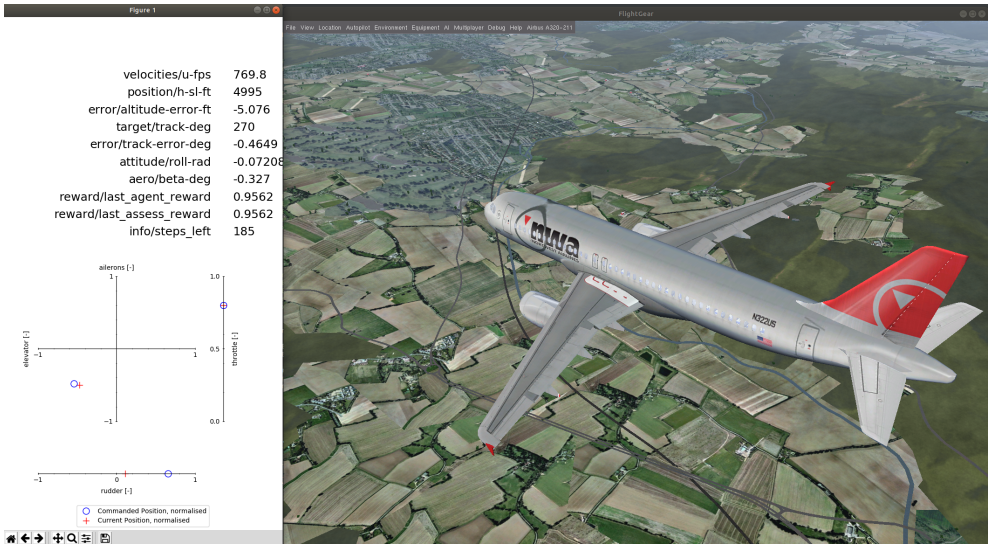

Figure 4-3: Visualisation of an Airbus A320 being controlled by an agent by streaming simulation data to FlightGear. A plot of aircraft control surface positions and state data is given by a `FigureVisualiser` on the left.

Internally, the `JsbSimEnv` delegates responsibility to its `Task` to input the action to the `Simulation`, run an appropriate number of simulation steps, retrieve a new state, and calculate reward. The `Task` encapsulates all aspects of the MDP except transition dynamics, allowing new flight control environments to be created by implementing a new `Task` subclass. The provided `Task`s at publication are `HeadingControlTask` and `TurnHeadingControlTask`.

Following the interaction step, any active `Visualiser` retrieves updated state information from the `Simulation` and displays it to the user (see Figure 4-3).

Reward

Gym-JSBSim implements several classes to allow for shaping rewards to be flexibly used. A researcher evaluating an agent in an environment with shaping rewards must sum the non-shaping reward R over the episode, while passing the agent shaping reward value $R + F$ for observations. To organise this process, a `Reward` class was implemented with methods to retrieve either value.

Responsibility for calculating rewards was delegated to an `Assessor` class, which was designed to calculate rewards based on a sum of normalised error complements over a range of variables being controlled (Equation 3.5). To

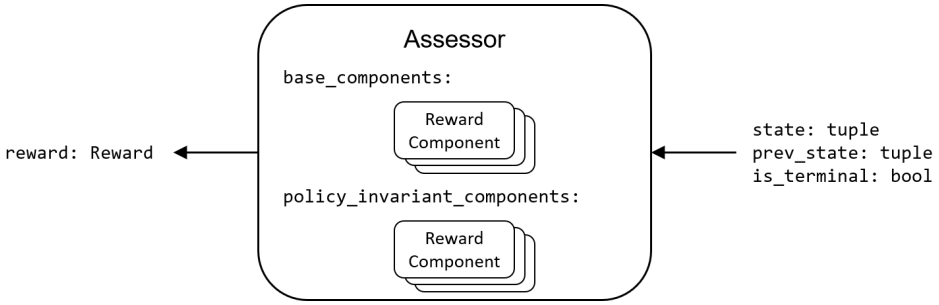

Figure 4-4: Schematic of the `Assessor` class design, showing its separate collections of `RewardComponent`s it stores for calculating `Reward`s from environment transitions.

allow flexible experimentation, a `RewardComponent` class was created which could be initialised with a state variable of interest, a target value and a scaling factor. The `RewardComponent` could then be called with a state, from which it would return a value for the normalised error complement.

Therefore an `Assessor` held two collections of `RewardComponent`s (see Figure 4-4) with which it could calculate the base reward R and policy invariant shaping reward F of a `Reward`. This arrangement was fully modular, and allowed reward functions controlling different state properties to be rapidly experimented with.

4.1.4 Testing

Gym-JSBSim was extensively tested with a combination of unit tests and integration tests (Table 4.1). The Agile practice of writing tests in parallel to production code was adopted, instead of treating testing as a separate phase at the end of the project. This provided confidence in the correctness of the codebase and was essential for identifying bugs introduced with new features or when old code was refactored. The automated test suite was supplemented with several manual tests covering aspects of visualisation which were impractical to automate.

The use of the third-party JSBSim library was identified as a risk early in the project; its Python API is relatively new and immature, its initialisation relies heavily on loading configuration files from disk, and its use of a different programming language, C++, makes it harder to debug problems. Following the methodology recommended by Martin (2008, pp.116–118), this risk was

Table 4.1: Gym-JSBSim automated test statistics; all tests are passing and cover the vast majority of the project's code.

Automated Tests	Number
Total	115
Passing	115
Coverage [% lines of code]	95 %

mitigated by using the wrapper class `Simulation` to handle all calls to JSBSim. The wrapper was then extensively unit tested to confirm the third-party API behaved as expected. This approach means that any behavioural differences in future revisions of JSBSim can be quickly detected and resolved by changes in `Simulation` without impact on the wider codebase.

4.1.5 Review

The Gym-JSBSim package met its high-level goals to provide configurable aircraft control RL environments based on a fast flight dynamics model and conforming to the widely-used OpenAI Gym interface (Brockman et al., 2016). Its integration with the FlightGear simulator provides an appealing 3D visualisation of aircraft under control. The iterative Agile processes and object-oriented design made the development process flexible and responsive to new functionality requirements that arose from experimental results and were well-suited to its use as a research tool.

For example, early experimental results showed undesirable agent behaviour which could be corrected through reward shaping. This prompted development of the modular system of `Assessor`s and `RewardComponent`s in the following sprint. Although developing and testing these general-purpose classes was a significant time investment for the project compared to hard-coding reward functions, it later enabled rapid experimentation with different reward functions and reward shaping methods that would not have been possible in the time available if each had to be individually written.

The package relies on interface inheritance over implementation inheritance for several key classes, such as `Task`s and `Assessor`s. This provides greater flexibility for future extensions, such as new control tasks, because subclasses are not coupled to implementation details in the base class.

The testing strategy evolved over the duration of the project to become

more efficient. Initially, unit tests were written for every function and method, including small helper methods not intended for public use. Since the code style followed the clean code principle of small functions (Martin, 2008, pp.34–37) this resulted in a large number of tests covering small implementation details. Although this was good for establishing the correctness of the code, it required significant work to maintain tests as they were broken by development iterations which changed implementation details. To avoid this, the test strategy shifted to cover the overall behaviour promised by public methods in unit tests and not individual helper functions.

Several lower-priority backlog items were incomplete at the end of the project. The largest item is to refactor the `HeadingControlTask` class, which holds too many responsibilities. For example, it defines the state and action space of the environment, sets initial episode conditions, determines whether a state is terminal, and may override rewards generated by an `Assessor` if the episode is terminal. These broad responsibilities should be delegated to improve the design's cohesion. For example, a new `Initialiser` class could set initial conditions, and responsibility for episode termination and termination rewards could be delegated to the `Assessor`, possibly encapsulating it within a new `Terminator` class.

Another outstanding item is to carry out performance profiling of the code. This would identify bottlenecks to target for optimisation to improve execution speed. However, the performance observed during the project was sufficient for a good pace of experimentation (requiring approximately 20 min to simulate 2000 episodes of 60 s of flight) and optimisation was not prioritised.

4.2 Experiment Tools

In addition to Gym-JSBSim, a smaller software package providing important functionality for agent training, hyperparameter search, and data collection was created. The overall objective was to automate agent training experiments and implement a randomised hyperparameter search, with the following requirements identified through user stories:

1 Create RL agents with specified hyperparameters using the TensorForce library

2 Create RL agents with hyperparameters randomly selected from a spec-

ified search space

3 Run agent-environment training episodes and receive the results

4 Run many agent-environment training episodes with specified hyperparameter values and environment

5 Train multiple agents concurrently

6 Save and restore agent results to/from disk

7 Save trained agents to disk

8 Restore trained agent from disk and visualise their learned behaviour in FlightGear

9 Provide compatibility with separate assessment and agent shaping rewards

The functionality was implemented using a modular, object-oriented design with classes:

- `AgentFactory`: creates TensorForce RL agents from input hyperparameters; generates random sets of hyperparameters from allowed values.
- `Experiment`: initiates training of agents in a pool of concurrent processes; stores the `AgentRecord`s produced; provides methods for saving/restoring self from disk
- `AgentRecord`: stores agent hyperparameter values and records of episode reward, times, and timesteps
- `ExperimentAnalyser`: utility class providing methods to analyse agent performance and visualise trained agents
- `JsbSimGym`, `JsbSimRunner`: small classes overriding methods in TensorForce's `Runner` and `OpenAIGym` classes to make them compatible with separate reward values for the agent and assessment (see Figure 4-5)

The user stories and CRC cards generated during the requirements gathering and design process are attached in Appendices A.2 and B.2 respectively.

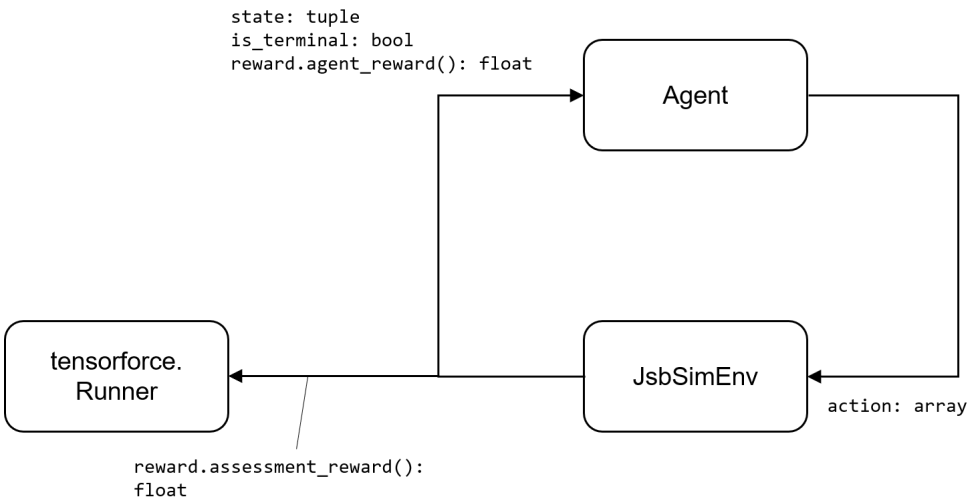

Figure 4-5: A modified `Runner` class provides the ability to extract non-shaping reward values from the data returned by a `JsbSimEnv` for assessment.

Table 4.2: Experiment tools package automated test statistics; all tests are passing and have good coverage of the codebase.

Automated Tests	Number
Total	45
Passing	45
Coverage [% lines of code]	90 %

This software package was also widely unit tested (Table 4.2). This was particularly helpful for detecting compatibility-breaking changes in the TensorForce package, which was under active development with frequent releases, and also isolated bugs in error-prone aspects of the implementation such as multiprocessing and storing data to disk.

Overall, the experimental tools developed for the project were extremely helpful for automating labour-intensive aspects of research, enabling rapid investigation of new ideas and theories.

Chapter 5

Experiments

Heading and altitude control of fixed wing aircraft has been formulated as an MDP and implemented in the Gym-JSBSim package. Deep RL agents will now be trained and evaluated in Gym-JSBSim environments over two experiments.

In the first experiment, the impact of providing additional shaping reward on agent performance and learning efficiency is investigated. The second experiment evaluates the ability of agents to learn to fly aircraft of different handling characteristics using the same hyperparameters.

5.1 Methods

5.1.1 Common Methods

Deep RL agent performance was evaluated on the two classes of environment implemented in Gym-JSBSim as formulated in Chapter 3:

- HeadingControl: the agent must maintain an aircraft's initial altitude and heading
- TurnHeadingControl: the agent must turn the aircraft to face a randomised target heading while maintaining its initial altitude

The environments were configured with a JSBSim timestep frequency (determining the accuracy of the simulation) of 60 Hz and an agent action frequency of 5 Hz. Episode duration was limited to 300 steps, corresponding to 60 s of flight. Engine controls were fixed at 0.8 throttle and 0.8 mixture.

Table 5.1: Reward function scaling factors, k, for calculating normalised errors from state (see Section 3.6.2). The scaling factor specifies at what error from the target the agent receives 0.5 reward or potential out of a maximum of 1.0.

State Variable	k	Unit
Altitude, h	150	ft
Heading, ψ	8	°
Roll, ϕ	8	°
Sideslip, β	3	°

Agents were implemented using the TensorForce (Schaarschmidt, Kuhnle and Fricke, 2017) library's implementation of the PPO algorithm (Schulman et al., 2017). Hyperparameter optimisation was performed using random search (Bergstra and Bengio, 2012) over 256 random configurations evaluated on the HeadingControl environment. The search space was populated with typical value ranges from other works applying PPO to complex continuous control tasks. The best hyperparameter configuration was determined from mean episode reward. The optimised hyperparameters and details of the full search space are provided in Appendix C.

All agents were evaluated by undiscounted cumulative reward, R, ignoring any shaping rewards. The reward scaling factors were fixed at the values provided in Table 5.1. Each cumulative reward was normalised in $[0, 1]$ by dividing by the maximum achievable episode reward (`episode_timesteps` $\times$ 1.0) for plotting. All experimental conditions were repeated seven times to account for randomness in the learning process.

5.1.2 Experiment 1: Reward Shaping

PPO agents were each trained in the HeadingControl and TurnHeadingControl environments in the Cessna 172P aircraft for 1×10^6 timesteps using the optimised hyperparameters. The reward function used in the environment was modified to one of three conditions, previously derived in Chapter 3:

1. **Standard**: the agent received only the base reward, R, for maintaining correct altitude and heading, with no additional shaping reward (Equation 3.4).

2. **Extra shaping**: the agent received the base reward plus a policy-

invariant shaping reward, $\hat{R} = R+F$, where F provides positive feedback for maintaining wings level and no sideslip (Equation 3.7).

3. **Extra sequential shaping**: the agent receives the base reward plus a sequential policy-invariant shaping reward, $\hat{R} = R + F'$, where F' provides additional feedback for maintaining no sideslip, and for keeping wings level when the aircraft is near its target heading (Equation 3.9).

5.1.3 Experiment 2: Dissimilar Aircraft Control

Experiment 2 investigated whether agents using the same hyperparameters could learn how to fly aircraft with different handling capabilities. PPO agents were trained in the HeadingControl and TurnHeadingControl environments with the 'Standard' reward function for 5×10^5 interaction steps using the optimised hyperparameters. The environment was modified to simulate flying one of three aircraft provided by the JSBSim flight dynamics model:

1. **Cessna 172P**: a single-engine light aircraft with mass[1] 680 kg
2. **McDonnell Douglas F-15**: a twin-engine fighter aircraft with mass[1] 1.3×10^4 kg
3. **Airbus A320**: a twin-engine jet airliner with mass[1] 5.0×10^4 kg

5.2 Results

Experiment 1 results are provided in Figure 5-1 and show learning occurred quickly in both environments (within 1×10^5 steps) before plateauing. In the HeadingControl environments, Agents learned policies that received approximately 0.5 of the maximum possible reward, with no noticeable difference when the reward function was varied. There was similarly no identifiable difference caused by changing reward functions in the TurnHeadingControl environments, and in this environment agents received approximately 0.35 of the total reward available following training.

Experiment 2 results are provided in Figure 5-2. They show agents learning policies of similar quality across all three aircraft tested.

[1] all values reflect empty mass sourced from JSBSim data files (Berndt and De Marco, 2009)

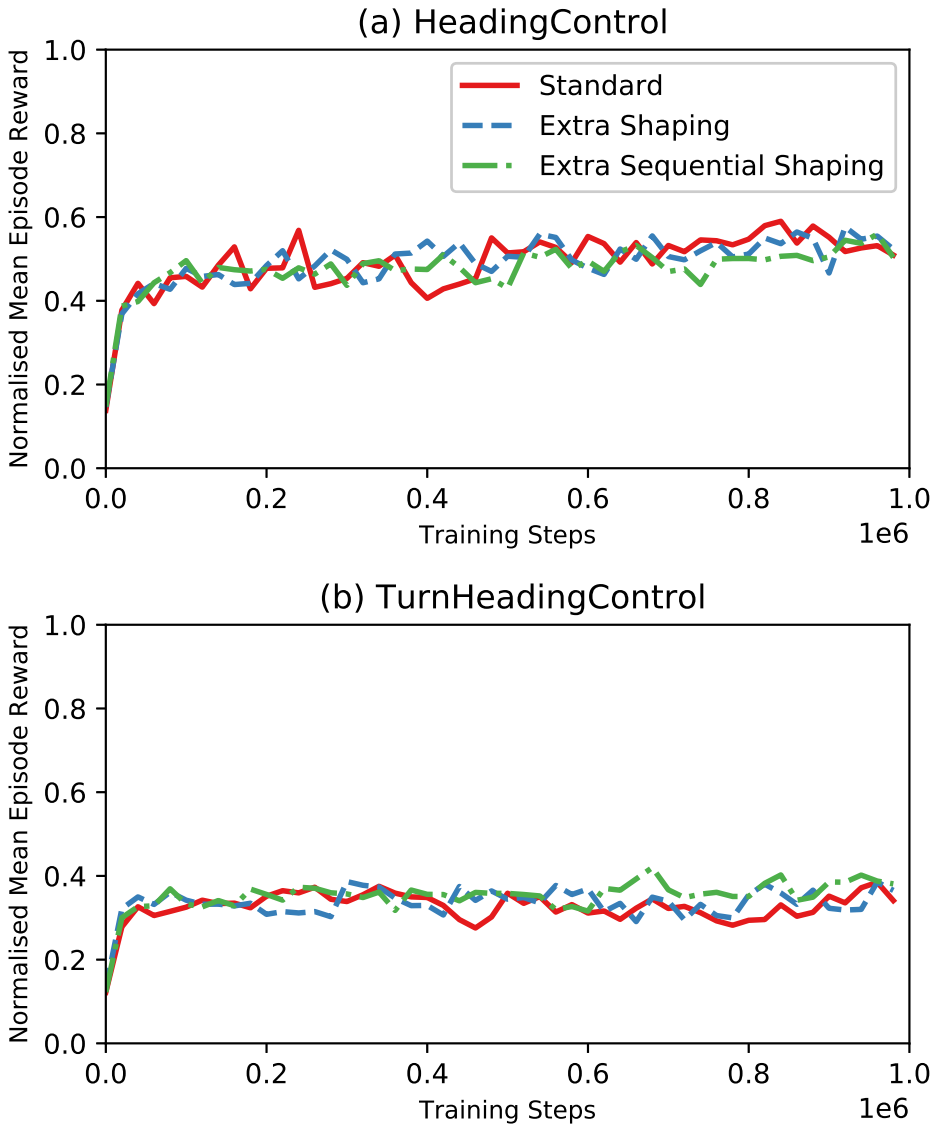

Figure 5-1: Learning curves for the **(a)** HeadingControl and **(b)** TurnHeadingControl environments with the Cessna 172P using different reward functions. Agent sample size $n = 7$ for all experiment conditions.

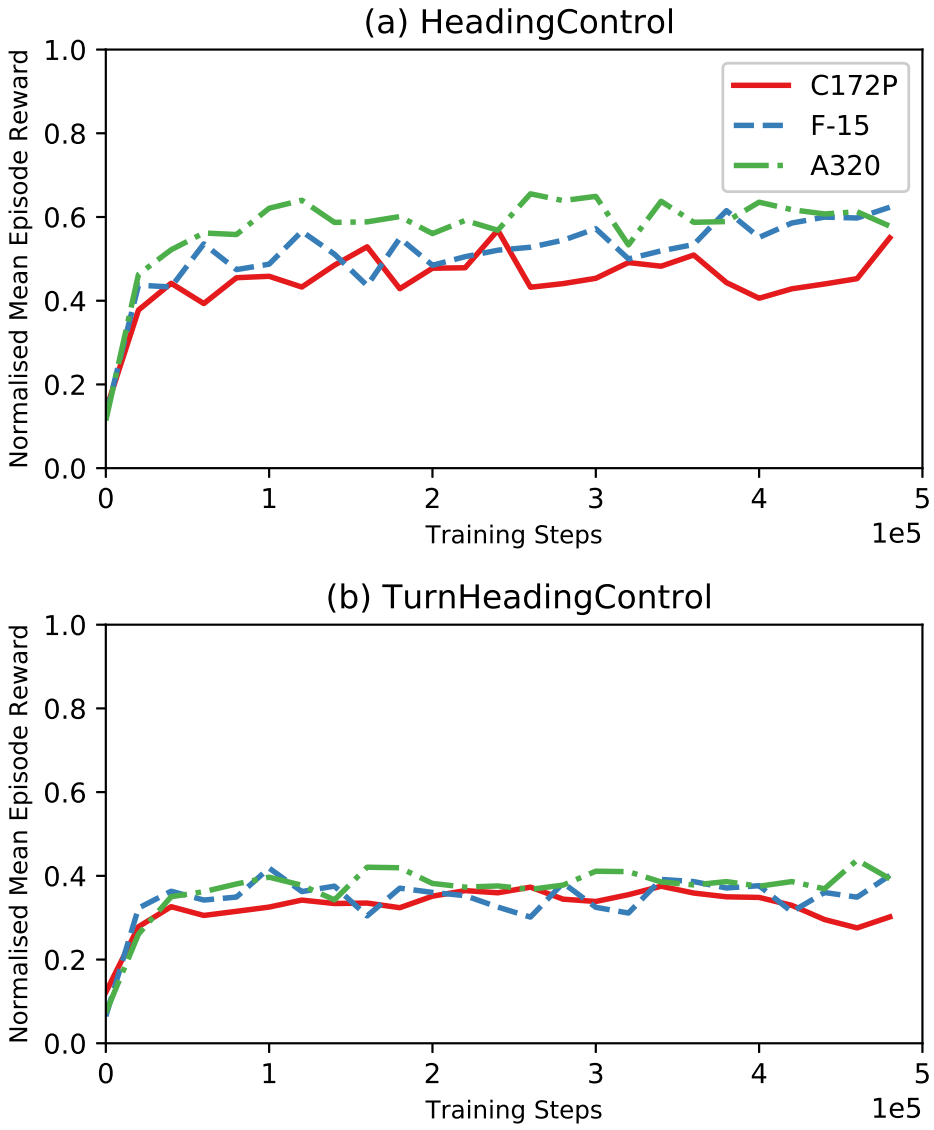

Figure 5-2: Learning curves for the **(a)** HeadingControl and **(b)** TurnHeadingControl environments with agents controlling the Cessna 172P, McDonnell Douglas F-15 and Airbus A320. Agent sample size $n = 7$ for all experiment conditions.

Following the experiments, trained agents from every experimental condition were loaded and visualised using Gym-JSBSim and FlightGear. The following qualitative, informal observations were made:

- No obvious behavioural differences were observed between agents receiving different shaping rewards.
- Agents made high frequency oscillating control inputs in one or more axes when near the target altitude or heading. Control surfaces were repeatedly moved in opposite directions every agent interaction (at a rate of 5 Hz). The net effect on aircraft motion appeared to cancel out.
- Almost all agents learned good altitude control, and the majority of their reward appeared to come from this. In one case, an agent in the HeadingControl environment was observed to have a policy with precisely maintained its heading, but with poor control of its altitude.
- Agents in the TurnHeadingControl environments failed to control their heading; agents either flew straight ahead without turning, or turned constantly in the same direction. No agents were observed to exit their turn upon reaching the target heading, which is the desired behaviour.

5.3 Discussion

Overall, the results and visualisations demonstrate agents learning to pilot aircraft over a controlled trajectory near the target altitude. This validates the MDPs for flight control formulated in this work and implemented in the Gym-JSBSim package.

There remains room for improvement. In the HeadingControl environments, agents averaged approximately half of the maximum achievable reward. This implies that agents controlled altitude and heading approximately with an error of one scaling factor. In the TurnHeadingControl task, agents learned to maintain their altitude but did not exhibit heading control as desired: they entered aircraft into turns, but did not level out when facing the correct direction. Overall, it appears agents struggle to learn to achieve both goals simultaneously.

Experiment 1 applied different shaping reward functions to try and counteract this deficiency. Disappointingly, no difference in learning curves or

performance was observed under the 'Extra Shaping' and 'Extra Sequential Shaping' conditions. One possibility is that the magnitude of the policy-invariant shaping rewards are too small relative to the base reward function to influence agent behaviour. The policy-invariant shaping function, F, could be increased by an arbitrary factor in future work.

Another avenue to improve agent performance is to increase exploration, so that agents are more likely to discover that exiting their turns on the desired heading is highly rewarding. Attempts were made to apply additional noise to agent actions, however no improvement was experienced (see supplementary results in Appendix D). Better results might be obtained from applying noise to the agent network parameters instead (Plappert et al., 2017) and possibly applying the off-policy DDPG algorithm, which together have recently shown superior results to PPO in robotics continuous control tasks (Kidzinski et al., 2018).

The oscillation observed in the agent's control inputs was interesting, and reminiscent of poorly-tuned PID controllers. It would be preferred if agents made smooth, gentle control inputs rather than the large, oscillating actions observed. This could be corrected by encoding this preference directly in the reward function, for example rewarding the agent for minimising forces experienced by the pilot. Additionally, the agent action frequency could be reduced, which would exclude policies which rely on high frequency oscillation and may replicate the improvements seen in Atari environments (Braylan et al., 2015).

Experiment 2 demonstrated that agents using the same hyperparameters were able to learn equally good policies across a variety of aircraft handling characteristics. In contrast to conventional PID controllers, which would require re-tuning when the aircraft dynamics are altered, RL agents are able to flexibly learn to pilot new aircraft.

Significant time and effort in this work was expended on agent hyperparameter optimisation. The random search used was simple and found effective parameter combinations. However, it is also a relatively inefficient and slow method, and is unlikely to have found optimal parameters. A better approach would be to use an informed optimisation method, which offer better hyperparameters in significantly less time (see *e.g.* Li et al., 2016).

Chapter 6

Conclusions

This work has formulated the heading and altitude control of fixed-wing aircraft as a Markov decision process. An appropriate reward function encoding the desired control behaviour was derived, and methods for providing additional reward shaping feedback to the agent were developed.

Investigation of fixed-wing flight control required a suitable simulation environment for RL agents to interact with. The Gym-JSBSim software package was designed and implemented for this purpose, providing configurable and fast environments for aircraft control using the popular OpenAI Gym interface. The package has been published under an open source license to enable further research in this area.

Deep RL agents using the PPO algorithm were evaluated in two classes of Gym-JSBSim environments of varying difficulty. The results showed that agents learned good policies for the simpler task requiring them to fly aircraft directly ahead. However, on the more challenging task where they had to learn to turn the aircraft to a specified heading, they failed to learn the more complex sequence of actions required to maximise reward. Additional shaping rewards and exploratory action noise were applied, but failed to correct the behavioural deficiency.

Further evaluations of deep RL agents were then carried out in a variety of different aircraft, with the results demonstrating that agents were able to flexibly learn good policies for each aircraft using the same hyperparameters.

6.1 Future Work

A variety of improvements and areas of investigation are identified, reflecting rapid advances in deep RL for continuous control tasks:

- Implement additional aircraft control tasks in Gym-JSBSim, *e.g.* waypoint navigation.
- Investigate use of network parameter noise for better exploration with the aim of improving performance on the TurnHeadingControl class of environments.
- Investigate increasing the magnitude of shaping rewards.
- Investigate use of alternate RL algorithms, such as DDPG, which may offer better performance.
- Investigate methods to reduce RL agent control oscillation, *e.g.* by reducing agent action frequency or encoding the preference in the reward function.

Bibliography

Andrychowicz, M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., Schneider, J., Sidor, S., Tobin, J., Welinder, P., Weng, L. and Zaremba, W., 2018. Learning dexterous in-hand manipulation. *CoRR* [Online], abs/1808.00177. 1808.00177, Available from: https://arxiv.org/abs/1808.00177.

Arulkumaran, K., Deisenroth, M.P., Brundage, M. and Bharath, A.A., 2017. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6), pp.26–38.

Beard, R.W., 2012. *Small Unmanned Aircraft: Theory and Practice*. Princeton, N.J.: Princeton University Press.

Bellman, R.E., 2015. *Adaptive Control Processes: A Guided Tour*. Princeton University Press.

Bergstra, J. and Bengio, Y., 2012. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(1), pp.281–305.

Berndt, J. and De Marco, A., 2009. Progress on and usage of the open source flight dynamics model software library, JSBSim. *Aiaa modeling and simulation technologies conference*. p.5699.

Braylan, A., Hollenbeck, M., Meyerson, E. and Miikkulainen, R., 2015. Frame skip is a powerful parameter for learning to play Atari. *Aaai-15 workshop on learning for general competency in video games*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W., 2016. OpenAI Gym. *CoRR* [Online], abs/1606.01540. 1606.01540, Available from: http://arxiv.org/abs/1606.01540.

Carrio, A., Sampedro, C., Rodriguez-Ramos, A. and Campoy, P., 2017. A review of deep learning methods and applications for unmanned aerial vehicles. *Journal of Sensors*, 2017.

Clark, J. and Amodei, D., 2016. *Faulty Reward Functions in the Wild* [Online]. OpenAI. Available from: `https://blog.openai.com/faulty-reward-functions/` [Accessed 7 August 2018].

Conde, R., Llata, J.R. and Torre-Ferrero, C., 2017. Time-varying formation controllers for unmanned aerial vehicles using deep reinforcement learning. *CoRR* [Online], abs/1706.01384. `1706.01384`, Available from: `http://arxiv.org/abs/1706.01384`.

Dewey, D., 2014. Reinforcement learning and the reward engineering principle. *AAAI Spring Symposium Series*.

Erdos, D. and Watkins, S.E., 2008. UAV autopilot integration and testing. *2008 IEEE Region 5 Conference*. pp.1–6.

Gimenes, R., Silva, D.C., Reis, L.P. and Oliveira, E., 2008. Using flight simulation environments with agent-controlled UAVs. *Autonomous Robot Systems and Competitions: Proceedings of the 8th Conference*.

Grześ, M., 2017. Reward shaping in episodic reinforcement learning. *Proceedings of the 16th Conference on Autonomous Agents and Multi-Agent Systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, AAMAS '17, pp.565–573.

Gu, S., Lillicrap, T., Sutskever, I. and Levine, S., 2016. Continuous deep Q-learning with model-based acceleration. *Proceedings of the 33rd International Conference on Machine Learning - Volume 48*. JMLR.org, ICML'16, pp.2829–2838.

Hammond, M., 2017. *Deep Reinforcement Learning in the Enterprise: Bridging the Gap from Games to Industry* [Online]. Bonsai. Available from: `https://www.youtube.com/watch?v=GOsUHlr4DKE` [Accessed 9 August 2018].

Hasselt, H.v., Guez, A. and Silver, D., 2016. Deep reinforcement learning with double Q-learning. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI Press, AAAI'16, pp.2094–2100.

Hayes, E., 2013. *Machine learning for intelligent control: Application of reinforcement learning techniques to the development of flight control systems for miniature UAV rotorcraft*. Thesis (MEng). University of Canterbury.

Heess, N., TB, D., Sriram, S., Lemmon, J., Merel, J., Wayne, G., Tassa, Y., Erez, T., Wang, Z., Eslami, S.M.A., Riedmiller, M.A. and Silver, D., 2017. Emergence of locomotion behaviours in rich environments. *CoRR* [Online], abs/1707.02286. `1707.02286`, Available from: `http://arxiv.org/abs/1707.02286`.

Hornik, K., 1991. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2), pp.251–257.

Kasnakolu, C., 2016. Investigation of multi-input multi-output robust control methods to handle parametric uncertainties in autopilot design. *PLOS ONE*, 11(10), pp.1–36.

Kidzinski, L., Mohanty, S.P., Ong, C.F., Huang, Z., Zhou, S., Pechenko, A., Stelmaszczyk, A., Jarosik, P., Pavlov, M., Kolesnikov, S., Plis, S.M., Chen, Z., Zhang, Z., Chen, J., Shi, J., Zheng, Z., Yuan, C., Lin, Z., Michalewski, H., Milos, P., Osinski, B., Melnik, A., Schilling, M., Ritter, H., Carroll, S.F., Hicks, J.L., Levine, S., Salathé, M. and Delp, S.L., 2018. Learning to run challenge solutions: Adapting reinforcement learning methods for neuromusculoskeletal environments. *CoRR* [Online], abs/1804.00361. `1804.00361`, Available from: `http://arxiv.org/abs/1804.00361`.

Kim, H.J., Jordan, M.I., Sastry, S. and Ng, A.Y., 2004. Autonomous helicopter flight via reinforcement learning. In: S. Thrun, L.K. Saul and B. Schölkopf, eds. *Advances in neural information processing systems 16*. MIT Press, pp.799–806.

Kimathi, S., Kang'ethe, S. and Kihato, P., 2017. Application of reinforcement learning in heading control of a fixed wing UAV using X-Plane platform. *International Journal of Scientific and Technology Research*, 6, pp.285–290.

Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. *CoRR* [Online], abs/1412.6980. `1412.6980`, Available from: `http://arxiv.org/abs/1412.6980`.

Konidaris, G., Osentoski, S. and Thomas, P., 2011. Value function approximation in reinforcement learning using the Fourier basis. *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence*. pp.380–385.

Lecun, Y., Bengio, Y. and Hinton, G., 2015. Deep learning. *Nature*, 521(7553), pp.436–444.

Li, L., Jamieson, K.G., DeSalvo, G., Rostamizadeh, A. and Talwalkar, A., 2016. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR* [Online], abs/1603.06560. `1603.06560`, Available from: `http://arxiv.org/abs/1603.06560`.

Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D., 2015. Continuous control with deep reinforcement learning. *CoRR* [Online], abs/1509.02971. `1509.02971`, Available from: `http://arxiv.org/abs/1509.02971`.

Marthi, B., 2007. Automatic shaping and decomposition of reward functions. *Proceedings of the 24th International Conference on Machine Learning*. New York, NY, USA: ACM, ICML '07, pp.601–608.

Martin, R.C., 2008. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR.

McLaughlin, B.D., Pollice, G. and West, D., 2006. *Head First Object-Oriented Analysis and Design*. O'Reilly Media, Inc.

Mnih, V., Badia, A.P., Mirza, M., Graves, A., Harley, T., Lillicrap, T.P., Silver, D. and Kavukcuoglu, K., 2016. Asynchronous methods for deep reinforcement learning. In: M.F. Balcan and K.Q. Weinberger, eds. *Proceedings of the 33rd International Conference on Machine Learning*. New York, New York, USA: PMLR, *Proceedings of Machine Learning Research*, vol. 48, pp.1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen,

S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D., 2015. Human-level control through deep reinforcement learning. *Nature*, 518, p.529.

Monaco, J.F., Ward, D.G. and Barto, A.G., 1998. Automated aircraft recovery via reinforcement learning: Initial experiments. In: M.I. Jordan, M.J. Kearns and S.A. Solla, eds. *Advances in neural information processing systems 10*. MIT Press, pp.1022–1028.

Ng, A.Y., Harada, D. and Russell, S.J., 1999. Policy invariance under reward transformations: Theory and application to reward shaping. *Proceedings of the sixteenth international conference on machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., ICML '99, pp.278–287.

OpenAI, 2017. OpenAI Dota 2 1v1 bot. `https://openai.com/the-international/`.

Pardo, F., Tavakoli, A., Levdik, V. and Kormushev, P., 2017. Time limits in Reinforcement Learning. *CoRR* [Online], abs/1712.00378. `1712.00378`, Available from: `http://arxiv.org/abs/1712.00378`.

Phansalkar, V.V. and Thathachar, M.A.L., 1995. Local and global optimization algorithms for generalized learning automata. *Neural Computation*, 7(5), pp.950–973.

Plappert, M., Houthooft, R., Dhariwal, P., Sidor, S., Chen, R.Y., Chen, X., Asfour, T., Abbeel, P. and Andrychowicz, M., 2017. Parameter space noise for exploration. *CoRR* [Online], abs/1706.01905. `1706.01905`, Available from: `http://arxiv.org/abs/1706.01905`.

Pollack, J.B. and Blair, A.D., 1996. Why did TD-Gammon work? *Proceedings of the 9th international conference on neural information processing systems*. Cambridge, MA, USA: MIT Press, NIPS'96, pp.10–16.

Polvara, R., Patacchiola, M., Sharma, S.K., Wan, J., Manning, A., Sutton, R. and Cangelosi, A., 2017. Autonomous quadrotor landing using deep reinforcement learning. *CoRR* [Online], abs/1709.03339. `1709.03339`, Available from: `http://arxiv.org/abs/1709.03339`.

Pratt, R., 2000. *Flight control systems: Practical issues in design and implementation*, Control, Robotics and Sensors Series. Institution of Electrical Engineers.

Randløv, J. and Alstrøm, P., 1998. Learning to drive a bicycle using reinforcement learning and shaping. *Proceedings of the fifteenth international conference on machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., ICML '98, pp.463–471.

Regan, K. and Boutilier, C., 2012. Regret-based reward elicitation for markov decision processes. *CoRR* [Online], abs/1205.2619. `1205.2619`, Available from: `http://arxiv.org/abs/1205.2619`.

Rummery, G.A. and Niranjan, M., 1994. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering.

Sanguineti, M. and Hlaváčková-Schindler, K., 1999. Some comparisons between linear approximation and approximation by neural networks. *Artificial neural nets and genetic algorithms*. Vienna: Springer Vienna, pp.172–177.

Schaarschmidt, M., Kuhnle, A. and Fricke, K., 2017. Tensorforce: A TensorFlow library for applied reinforcement learning [Online]. Web page. Available from: `https://github.com/reinforceio/tensorforce`.

Schaul, T., Quan, J., Antonoglou, I. and Silver, D., 2015. Prioritized experience replay. *CoRR* [Online], abs/1511.05952. `1511.05952`, Available from: `http://arxiv.org/abs/1511.05952`.

Schulman, J., Moritz, P., Levine, S., Jordan, M.I. and Abbeel, P., 2015. High-dimensional continuous control using generalized advantage estimation. *CoRR* [Online], abs/1506.02438. `1506.02438`, Available from: `http://arxiv.org/abs/1506.02438`.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. *CoRR* [Online], abs/1707.06347. `1707.06347`, Available from: `http://arxiv.org/abs/1707.06347`.

Silver, D., 2015. *Deep reinforcement learning*. Multidisciplinary Conference on Reinforcement Learning and Decision Making.

Silver, D., 2016. *Tutorial: Deep reinforcement learning.* International Conference on Machine Learning.

Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Driessche, G. van den, Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D., 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529, p.484.

Sommerville, I., 2015. *Software Engineering.* 10th ed. London, UK: Pearson.

Story, R., 2017. *Writing Great Reward Functions* [Online]. Bonsai. Available from: `https://www.youtube.com/watch?v=0R3PnJEisqk` [Accessed 7 August 2018].

Sutton, R.S. and Barto, A.G., 1998. *Introduction to reinforcement learning.* 1st ed. Cambridge, MA, USA: MIT Press.

Sutton, R.S. and Barto, A.G., 2018. *Introduction to reinforcement learning.* 2nd ed. MIT Press. Unpublished work in progress.

Sutton, R.S., McAllester, D., Singh, S. and Mansour, Y., 1999. Policy gradient methods for reinforcement learning with function approximation. *Proceedings of the 12th International Conference on Neural Information Processing Systems.* Cambridge, MA, USA: MIT Press, NIPS'99, pp.1057–1063.

Tesauro, G., 1995. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38(3), pp.58–68.

Tsitsiklis, J.N. and Roy, B.V., 1997. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5), pp.674–690.

Uhlenbeck, G.E. and Ornstein, L.S., 1930. On the theory of the brownian motion. *Phys. Rev.* [Online], 36, pp.823–841. Available from: `https://doi.org/10.1103/PhysRev.36.823`.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M. and De Freitas, N., 2016. Dueling network architectures for deep reinforcement learn-

ing. *Proceedings of the 33rd International Conference on Machine Learning*. JMLR.org, ICML'16, pp.1995–2003.

Watkins, C.J. and Dayan, P., 1992. Technical note: Q-learning. *Machine Learning*, 8(3), pp.279–292.

Weaver, L. and Tao, N., 2001. The optimal reward baseline for gradient-based reinforcement learning. *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., UAI'01, pp.538–545.

Wiewiora, E., 2003. Potential-based shaping and q-value initialization are equivalent. *J. Artif. Int. Res.*, 19(1), pp.205–208.

Williams, R.J., 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3), pp.229–256.

Appendix A

User Stories

A.1 Gym-JSBSim User Stories

ID	as a...	I want to...	so that...	Conditions of Satisfaction
1	researcher	creates JSBSim instances in Python	I can interact with the simulation	• JSBSim instance is created • JSBSim directories are configured
2	researcher	load an aircraft into a JSBSim instance	I can simulate different aircraft	• Aircraft configs can be loaded by name of aircraft • Helpful error message displayed if aircraft config file not found
3	researcher	set an aircraft's initial conditions in the JSBSim simulation	I can start the episode in the state I want	• Every simulation variable is set to the value specified.
4	researcher	extract aircraft state information from JSBSim	I can pass Agents state observations	• Can call JSBSim with collection of desired variables and receive values back • Throws exception if requested variable does not exist
5	researcher	pass commands to JSBSim	aircraft can be controlled	• Can pass JSBSim collection of commands • JSBSim correctly updates aircraft in response to actions
6	researcher	run a simulation while making state observations and passing commands at regular intervals	an RL agent can interact with the simulation as an MDP	• Simulation timestep size between observations is configurable • Simulation conforms to the OpenAI Gym Environment interface
7	agent	query a JSBSim environment for its state and action space	I can configure my internal model for the correct number of variables	• A JsbSimEnv object can be queried for its state/action space, including high and low limits

ID	as a...	I want to...	so that...	Conditions of Satisfaction
				• Uses OpenAI Gym Spaces class
8	researcher	Implement different control tasks as JSBSim environments using a modular framework	Agents can be tested on different control tasks	• Task module specifies states/actions/rewards • Task module provides initial conditions each episode reset • Can instantiate environments implementing arbitrary tasks
9	researcher	visualise aircraft state	I can understand what is going on	• Aircraft position and velocities are visible • Visualisation starts and closes in response to Env `render()` method
10	researcher	visualise the aircraft current control surface positions and commanded positions	I can understand how the aircraft is being controlled	• Aircraft aileron, elevator, rudder, throttle positions and commanded positions are visible in the environment render. • Control positions shown as a subplot alongside aircraft state
11	researcher	provide a Task for flying in a straight line on a constant heading	I can investigate agent performance on a relatively simple task	• Implements the `Task` interface • an environment using this task can be created using `gym.make()` • Initial state has aircraft flying on the target heading • Task is tested on a simple random agent
12	researcher	implement an agent that makes	I can use it for functional tests and	• Agent makes random actions within

ID	as a...	I want to...	so that...	Conditions of Satisfaction
		random actions	sanity checks	the permitted range of the action space • Agent random seed can be specified
13	researcher	implement an agent that makes constant actions	I can use it for functional tests and sanity checks	• Agent always returns the same action, which is within the permitted range of the action space (e.g. midpoint)
14	researcher	visualise the aircraft graphically in a flight simulator	Viewers can appreciate the agent	• FlightGear is launched by the `render()` method • FlightGear displays the aircraft • FlightGear is closed by the `Environment.close()` method
15	researcher	visualise the aircraft's control surface position and commanded position alongside FlightGear	I can clearly observe the agent's actions	• Rendering the episode in FlightGear launches a figure with control surface positions • Figure closes on `Environment.close()`
16	researcher	see task-related state and reward data on environment visualisations	I can understand how 'good' the agent's behaviour is and how much reward it is generating	• visualisations display reward data • visualisations show print-outs of relevant state data
17	researcher	use reward shaping in the environments	its effects on agent learning speed can be investigated	• reward shaping is an environment option (True/False) • reward shaped with a potential-based function

ID	as a...	I want to...	so that...	Conditions of Satisfaction
				• Initial state has aircraft on a random heading with a random target heading
18	researcher	provide a more challenging control Task for flying in a straight line on a constant heading requiring turning the aircraft	I can investigate agent performance in a more difficult task	• Implements the `Task` interface • an environment using this task can be created using `gym.make()`
19	researcher	separately track shaping rewards and assessment rewards resulting from agent interactions	I can consistently compare the performance of agents using different reward shaping functions	• An agent step produces two reward values, one to pass to the agent and one for assessment of performance. • Implementation maintains compatibility with OpenAI Gym interface
20	researcher	Flexibly define reward functions based on error between current state value(s) and desired value(s)	I can experiment with different reward function designs	• Reward functions can be specified by providing the state variable of interest and its target value • Target value can be provided as a constant, or extracted dynamically from state • Reward values are normalised in a small interval to help training
21	researcher	Flexibly define reward functions based on state value errors with sequential dependencies between state values	I can experiment with reward functions with sequential goals	• Reward functions can be specified by providing a state variable of interest, its target values, and the same details for one or more variables which it is sequentially dependent upon

A.2 Experimental Tools User Stories

ID	as a...	I want to...	so that...	Conditions of Satisfaction
1	researcher	produce a TensorForce agent configured with a set of input hyperparameter values	I can evaluate a range of agents with an automated hyperparameter search	• an appropriately configured TensorForce Agent is returned
2	researcher	produce a TensorForce PPO Agent configured with a random permutation of hyperparameter values	I can run a random hyperparameter search	• search space for each hyperparameter can be defined • PPOAgent object is correctly configured
3	researcher	run many episodes with a TensorForce Agent in an environment and receive all results	I can evaluate agent performance	• trained Agent and collection of episode rewards are returned
4	researcher	run many episodes with a specified set of environment parameters and Agent hyperparameters	I can run an automated hyperparamer search	• Agent and Environment correctly configure themselves according to input • trained Agent and collection of episode rewards are recorded
5	researcher	train multiple agents concurrently	I can evaluate agent performance quickly	• Experiment runs with simultaneous training processes • All training results and agents are collected and stored • Concurrency implementation is compatible with JSBSim (which cannot run multiple instances on single process)
6	researcher	save and restore agent training results to/from disk	I have a persistent record of results for analysis and plotting	• all AgentRecord data are saved / restorable

ID	as a...	I want to...	so that...	Conditions of Satisfaction
				• appropriate error handling if file already exists / does not exist
7	researcher	save the best Agent(s) during evaluation and re-load them later	I can observe trained agent behaviour or train them over additional episodes	• Agent is saved in trained condition • Agent can be reloaded to trained state
8	researcher	load a trained agent and visualise its behaviour with learning and exploration disabled	I can observe trained agent behaviour	• Agent is loaded with trained model • Aircraft is visualised in FlightGear • Agent does not learn during interactions • Agent exploration is disabled
9	researcher	Pass a shaping reward to the agent, while storing a separate assessment reward for evaluation of performance	I can evaluate the performance of shaping reward functions against non-shaping reward functions	• Agent is passed shaping reward values • Cumulative episode rewards are calculated as the sum of assessment rewards

Appendix B

Class Responsibility Collaboration Cards

B.1 Gym-JSBSim CRC Cards

JsbSimEnv	gym.Env
• provde OpenAI Gym Env environment interface to RL agent • manage Simulation object • manage Visualiser objects if required • tidy up all environment objects on `.close()`	• Simulation • TaskModule • Agent • FlightGearVisualiser

Simulation	
• instantiate and configure JSBSim instance (FGFDMExec) • get and set simulation variables in JSBSim • run JSBSim simulation steps	• FGFDMExec • JsbSimEnv

<<Interface>> **Task**	
• retrieve / calculate state and termination every step • specify action and state variables • calculate initial agent state on environment reset • specify how to render the environment	• Simulation • JsbSimInstance

Assessor	
• hold RewardComponents corresponding to base and policy-invariant rewards • calculate `Rewards` from state transitions using RewardComponents	• RewardComponent

SequentialAssessor	Assessor
as per Assessor, and: • apply sequential dependencies to shaping reward values	• RewardComponent

Reward	
• encapsulate base reward and policy invariant shaping reward values • return an assessment reward or agent reward scalar	• Runner • Agent

RewardComponent	
• calculate reward values from state transitions • calculate state potentials	• Assessor

FigureVisualiser	
• manage creation, configuration and deletion of a multi-plot figure showing control surface state • update plots with latest state data when called	• matplotlib.pyplot.Figure

FlightGearVisualiser	
• manage creation and shutdown of a subprocess running the FlightGear simulator configured to visualise data stream from a Simulator object • manage a FigureVisualiser for rendering control surface actions alongside FlightGear	• subprocess.Popen • FigureVisualiser

Aircraft	
• store data relating to one aircraft type: JSBSim id, FlightGear id, cruise speed, *etc.*	

Property	
• store string literal used to access simulation property from JSBSim	• Simulation

BoundedProperty	
• Store string literal used to access simulation property form JSBSim • Store min/max values of bounded properties	• Simulation

B.2 Experimental Tools CRC Cards

Experiment	
• inits and trains Agents in specified conditions using Runner • creates and stores AgentRecords • saves ExperimentResults to disk • loads specified Experiments from disk according to ID	• AgentRecord • tensorforce.Agent • tensorforce.OpenAIGym • tensorforce.Runner

ExperimentAnalyser	
• analyses results and returns top AgentRecords according to heuristic function • plots learning curves from Experiment • visualises trained agents acting in environment	• Experiment • AgentRecord • tensorforce.OpenAIGym • tensorforce.Runner • matplotlib.pyplot.Figure

AgentRecord	
• stores agent initialisation kwargs • stores training reward history extracted from Runner • stores relative path to agent TensorFlow checkpoint • saves, retrieves and deletes agent network checkpoint on disk	• tensorforce.Agent • Runner

AgentFactory	
• creates Agent objects with random permutation of hyperparameters • creates Agent objects with specified hyperparameters • contains lists of values that hyperparameters may take	• tensorforce.Agent • tensorforce.OpenAIGym

Appendix C

Agent Hyperparameters

Table C.1: The PPO hyperparameter search space used for optimising agents in this work.

Hyperparameter	Search Space
Network hidden layers	$\{2, 3, 4\}$
Network nodes per layer	$\{32, 64, 128\}$
Network activation function	{tanh}
Network optimiser	{Adam[a]}
Network learning rate	$\{3 \times 10^{-5}, 1 \times 10^{-4}, 3 \times 10^{-4}, 1 \times 10^{-3}\}$
Network optimisation steps	$\{3, 5, 10\}$
Baseline hidden layers[b]	$\{2, 3, 4\}$
Baseline learning rate	$\{3 \times 10^{-5}, 1 \times 10^{-4}, 3 \times 10^{-4}, 1 \times 10^{-3}\}$
Baseline optimisation steps	$\{3, 5, 10\}$
Batch size [episodes]	$\{1, 4, 16\}$
Sub-sampling fraction	$\{0.05, 0.15, 0.30\}$
GAE parameter[c] (λ)	$\{0.8, 0.95, 0.99\}$
Clipping parameter (ϵ)	$\{0.05, 0.10, 0.20\}$
Entropy coefficient	$\{0, 1 \times 10^{-3}, 1 \times 10^{-2}, 1 \times 10^{-1}\}$
Discount factor (γ)	$\{1.0\}$
States normalisation	{`none`, `running_standardise`[d]}

[a] Kingma and Ba (2014)

[b] baseline network nodes per layer, activation function and optimiser was kept equal to that of the main network

[c] Generalised advantage estimation, see Schulman et al. (2015)

[d] `running_standardise` transforms state values to their deviation from the mean; a running mean and standard deviation for each state variable is maintained by the agent from all past observations

Table C.2: The optimised PPO agent hyperparameters used in this work following a randomised search over 256 configurations.

Hyperparameter	Optimised Value
Network hidden layers	2
Network nodes per layer	64
Network activation function	tanh
Network optimiser	Adam
Network learning rate	3×10^{-4}
Network optimisation steps	10
Baseline hidden layers	3
Baseline learning rate	1×10^{-3}
Baseline optimisation steps	3
Batch size [episodes]	1
Sub-sampling fraction	0.30
GAE parameter (λ)	0.99
Clipping parameter (ϵ)	0.10
Entropy coefficient	1×10^{-3}
Discount factor (γ)	1.0
States normalisation	`running_standardise`

Appendix D

Supplementary Results

D.1 Motivation

PPO agents trained in the TurnHeadingControl environment were observed to learn to maintain their altitude at target, but not to turn to face the target heading. It was hoped that adding additional noise to agent actions would improve exploration and allow agents to learn better policies. An Ornstein-Uhlenbeck (OU) process (Uhlenbeck and Ornstein, 1930) was used to generate temporally correlated noise, which is more suitable for exploration in continuous control problems (Lillicrap et al., 2015).

D.2 Methods

Agents were trained in the TurnHeadingControl environment with a Standard reward function (see Section 5.1.2) flying the Cessna 172P. Three experimental conditions were tested:

- **No extra exploration**: no noise is applied to agent actions
- **Low exploration**: OU process noise is added to agent actions with parameters $\sigma = 0.15$, $\mu = 0.0$, $\theta = 0.10$
- **High exploration**: OU process noise with higher variance is added to agent actions with parameters $\sigma = 0.30$, $\mu = 0.0$, $\theta = 0.10$

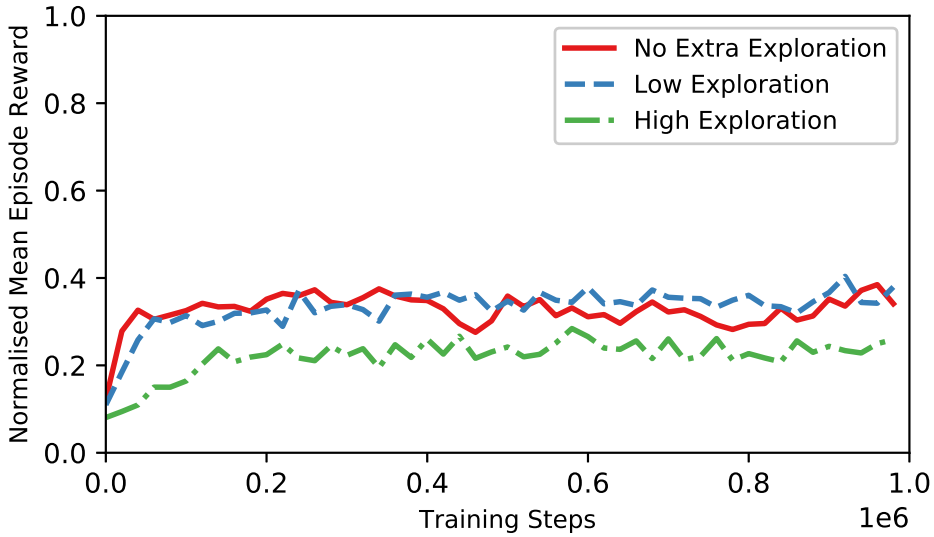

Figure D-1: Learning curves for the TurnHeadingControl environment with the Cessna 172P and Standard reward function. Two of the experiment conditions add additional Ornstein-Uhlenbeck process noise to the agent's actions to cause exploration and improve the agent's policy. Agent sample size $n = 7$ for all experiment conditions.

D.3 Results

Agent learning curves are provided in Figure D-1. The addition of exploration noise did not result in agents learning better policies, and visualisations of agent control performance showed that they continued to maintain altitude without turning to face the target heading. Mean episode reward declined in the case of 'high exploration', which can be attributed to the higher noise causing the agent to make poor actions.